Escuela Politécnica Superior

Estudio sobre el uso del deep learning en robots de bajo coste para la navegación en interiores

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Javier García Sigüenza

Tutor/es:
Sergio Orts Escolano
Félix Escalona Moncholi

Septiembre 2019

Universitat d'Alacant
Universidad de Alicante

"Somos como enanos sobre los hombros de gigantes, podemos ver más allá y más lejos que ellos, no por la virtud de una aguda visión de nuestra parte, o cualquier otra distinción física, sino porque somos elevados por su gigantesco tamaño."

Bernardo de Chartres

Agradecimientos

A mi familia por haberme apoyado durante la realización mis estudios universitarios.

A mis tutores por el interés mostrado y la ayuda dada.

A mi pareja por el apoyo recibido durante la realización de este trabajo.

Resumen

El deep learning ha abierto un mundo de posibilidades para la robótica. En este trabajo se ha decidido explorarlo con el objetivo de desarrollar un sistema que permita ayudar a un robot a reconocer su entorno y actuar en relación a este, para así mejorar su navegación en interiores. Se tiene como objetivo su implementación en un hardware de coste moderado, por lo que todo el trabajo se ha realizado buscando obtener un equilibrio entre rendimiento y precisión.

Para conseguir este objetivo, en primer lugar, se ha realizado un estudio sobre los fundamentos teóricos del aprendizaje automático hasta llegar al deep learning, para así poder conocer las diferentes alternativas existentes actualmente para resolver el problema propuesto. Tras ello, se han explorado estas alternativas y se ha buscado la más optima.

Finalmente, se ha realizado la implementación del sistema siguiendo las bases teóricas previamente explicadas. Para ello, se ha trabajado con redes neuronales, se ha creado el dataset necesario, se han realizado pruebas de rendimiento y se ha creado una simulación.

Índice

Lista de figuras

Lista de tablas

Lista de fórmulas

1. Introducción

1.1. Motivación

La motivación para realizar este trabajo ha sido la de poder crear un sistema que sirva para ayudar a los robots a poder reconocer su entorno, para así poder navegar en este de forma dinámica, permitiéndoles adaptarse a los diferentes elementos que haya a su alrededor y soportando cambios en estos.

La importancia de este sistema para los robots reside en que les ayuda a pasar de ser elementos estáticos y condicionados a unas circunstancias muy controladas, a tener la capacidad de relacionarse con el entorno y reaccionar ante este. Para este propósito se ha visto en el campo de la IA una gran oportunidad, ya que esta tecnología no permite únicamente el reconocimiento de elementos, sino también tener una mayor adaptabilidad de la que se obtenía con técnicas de visión artificial tradicionales.

Si bien el inicio del estudio del campo de la IA se remonta a la década de los años 40 del siglo pasado, ha sido en estos últimos años donde más ha crecido y un mayor uso práctico está teniendo, abriéndose a nuevas utilidades y aplicaciones. Por suerte, este aumento en su uso e interés ha ayudado al desarrollo del deep learning, perteneciente al campo de la IA, en el que se ha hecho un gran esfuerzo por seguir la filosofía Open Source. Por ello actualmente es una tecnología accesible a todos los interesados en el campo, existiendo una gran cantidad de información y herramientas a nuestra disposición, que utilizaremos para el desarrollo de este trabajo.

Este proyecto se centrará en buscar las mejores herramientas, en relación al rendimiento y precisión, para conseguir diseñar un sistema que ayude a la navegación en interiores de robots de bajo coste. Para ello habrá que tener en cuenta en todo momento el hardware necesario para conseguir realizar uso de esta tecnología a un coste moderado.

1.2. Objetivos

El objetivo de este estudio es el de desarrollar un sistema para la navegación de robots de bajo coste en interiores utilizando técnicas de deep learning. Para ello habrá que desarrollar una red neuronal que pueda ser ejecutada en un hardware que en su conjunto tenga un precio inferior a 1000€ y sea capaz de hacer funcionar la red a al menos 30 fps, para poder

ser utilizada en aplicaciones en tiempo real. Además, realizar una serie de pruebas para comprobar los resultados obtenidos, así como su rendimiento.

Para conseguir este objetivo hay una serie de puntos que hay que explorar y aprender:

- Entender los fundamentos teóricos del machine learning y del deep learning.
- Explorar las diferentes alternativas que existen dentro del deep learning para poder realizar una navegación en interiores.
- Elegir la red que más se adapte a nuestras necesidades y entrenarla para obtener resultados en un conjunto de datos que nos sirva de prueba.
- Calcular la precisión obtenida en el dataset, así como comprobar los resultados de estos fuera de este conjunto de datos.
- Probar la red obtenida en un hardware apto para el objetivo del estudio y comprobar su rendimiento.
- Realizar la simulación de un sistema que ayude a la navegación en interiores haciendo uso del deep learning.

Hay que resaltar que la idea con la que se ha planteado este proyecto es la de realizar un sistema para detectar las zonas por las que es posible navegar, por la que no y los posibles obstáculos. De esta manera se consigue tener un entendimiento general del entorno en el que se encuentra el robot, a partir de la entrada de una cámara, permitiendo la interacción con los elementos a su alrededor.

A pesar de ello, se tiene en cuenta que este es un sistema sencillo, que puede necesitar otros mecanismos auxiliares ante problemas más complejos. Esto quiere decir que utilizándolo se podría realizar, por ejemplo, el control de un robot que recorriese una nave de punta a punta siendo capaz de detectar obstáculos ante los que reacciones, o en su lugar, pararse y enviar una señal de emergencia. Este podría distinguir el final del trayecto mediante elementos mecánicos, u otros elementos visuales, como la detección de un código QR. Aunque para problemas de navegación más complejos, que necesitarán un control de tráfico o unos trayectos variables, harían falta sistemas auxiliares, como puede ser la odometría mecánica o visual, para ser capaz de determinar la posición del robot en un mapa dado.

En conclusión, el sistema que se expondrá en este trabajo tiene como objetivo ser capaz de ayudar por sí mismo a la navegación en interiores, pero pudiendo necesitar sistemas auxiliares que lo complementen dependiendo del problema concreto a solucionar.

1.3. Contexto

La investigación sobre la navegación de robots es un campo que ha tenido un gran recorrido. Inicialmente, los sistemas de navegación más populares se basaban en el uso de sensores láser, habiendo trabajos especializados en optimizar su uso para entornos concretos y mejorar así su navegación [1]. Además, ya antes de la popularización del deep learning se trabajaba en la navegación de robots a partir de imágenes usando visión artificial tradicional, por ejemplo, en base a la obtención de descriptores con el algoritmo SIFT [2].

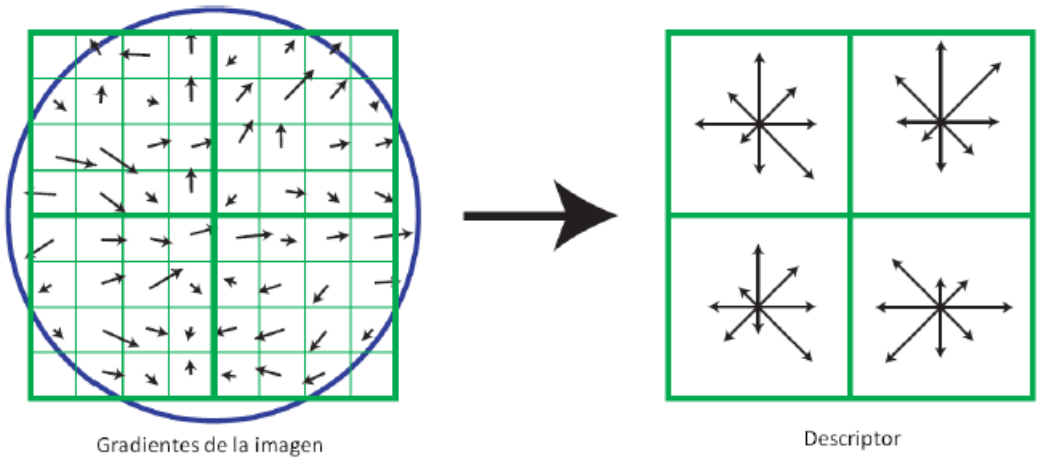


Figura 1. Proceso de creación de un descriptor 2x2 de SIFT a partir de regiones de 8x8 píxeles [6].

SIFT se basa en intentar obtener características de las imágenes que sean en gran medida invariantes a la escala, a la rotación, a los cambios de iluminación, el ruido y cambios moderados de la perspectiva y la pose de la imagen. Para ello se realiza un análisis por zonas alrededor de un punto seleccionado. Este análisis se basa en realizar un histograma que se calcula a partir de los valores de orientación, obtenidos utilizando los valores de intensidad de los pixeles, y magnitud de la región seleccionada.

Como explicación más simplificada e ilustrativa, los descriptores se les podría interpretar simplemente como elementos característicos y reconocibles de la imagen para el algoritmo, como pueden ser el cambio de texturas y de intensidades de los pixeles.

En el caso de la figura 2, valdría como característica para el algoritmo, por ejemplo, las porciones de la imagen indicadas por la letra F y por la E, ya que son elementos que se pueden localizar en la imagen de manera inequívoca. En cambio, para reconocer este edificio la porción de la imagen indicada por la letra A no valdría, ya que no representa ningún elemento característico del edificio.

Figura 2. Simplificación de las características de una imagen [7].

El problema de SIFT es que es muy dependiente del entorno, es decir, su correcto funcionamiento está muy ligado a ser usado en un entorno controlado. Es por ello por lo que este algoritmo ha sido descartado para su uso en este trabajo. En su lugar se explorará el uso de técnicas de deep learning como principal medio para conseguir el objetivo expuesto. Esto es debido a la consideración de que con esta tecnología se puede llegar a obtener un resultado mucho mejor, en lo referente al objetivo de este trabajo, que con los métodos expuestos anteriormente.

1.4. Estructura

La estructura de este proyecto se adapta a su proceso de desarrollo, iniciando el trabajo con una explicación de las bases teóricas sobre las que se fundamenta el deep learning, seguido por su estado del arte y la metodología del estudio. Tras esto se comienza a tratar la

implementación elegida, los resultados conseguidos, la simulación realizada y las conclusiones obtenidas a partir de estos datos.

Más detalladamente las estructura es:

- Capítulo 1: Introducción al problema mostrando la motivación, los objetivos y el contexto del trabajo.
- Capítulo 2: Bases del deep learning. En este capítulo se realiza una introducción general a los cimientos del deep learning, tras lo que se comienza a realizar una descripción más detallada de los elementos más importante a nivel teórico para poder realizar este trabajo.
- Capítulo 3: Estado del arte. Se explorará el estado del arte del deep learning, dentro del campo de las CNNs que realizan una segmentación semántica. Explorando diferentes redes y mostrando sus diferencias y avances. Finalmente se establecerá que red se considera la más adecuada de todas las estudiadas.
- Capítulo 4: Metodología y herramientas utilizada para realizar este trabajo.
- Capítulo 5: Exposición del trabajo práctico realizado a lo largo de la realización de este proyecto, así como los resultados obtenidos y un análisis de estos.
- Capítulo 6: Simulación en Unreal Engine 4 utilizando el plugin AirSim y análisis de los resultados conseguidos en esta.
- Capítulo 7: Conclusiones. En este apartado se muestran las diferentes conclusiones obtenidas en base a este trabajo.

2. Fundamentos del deep learning

2.1. Introducción al machine learning

Para introducir al deep learning lo más correcto es definir antes el machine learning, o aprendizaje automático en español, ya que este primero se puede entender como un subconjunto del segundo y a su vez ambos se encuentran dentro del campo de la IA. El machine learning supone un salto respecto a los sistemas de IA basados en reglas, es decir, aquellos basados en sentencias condicionales. Esto es debido a que este no necesita estar basado en una interpretación realizada por las personas, sino que esta se obtiene a partir de los datos proporcionados al propio sistema, buscando conseguir una función que se ajuste de la forma más precisa posible a los datos de entrada, obteniendo el mejor resultado posible.

Para un mayor entendimiento del funcionamiento y flujo de trabajo del machine learning se puede usar el esquema mostrado en la figura 3 como referencia.

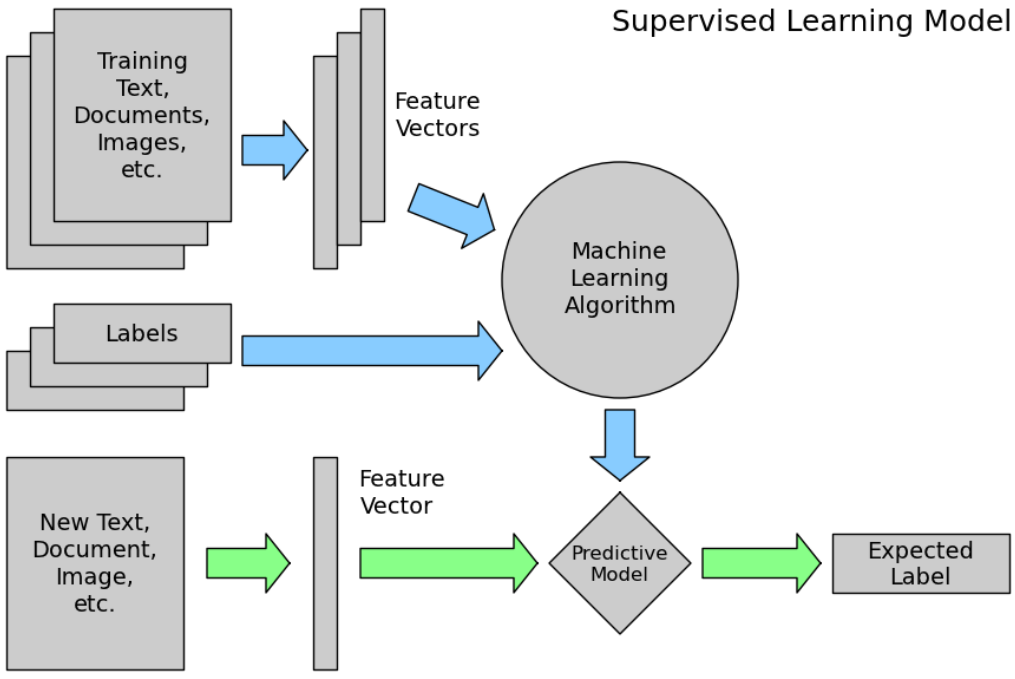


Figura 3. Esquema sobre el entrenamiento y la predicción de datos en machine learning [8].

Primero, a partir de unos datos dados se extraen los vectores de características (indicado en la figura 3 como “features”). Estos son aquellos valores que consideramos característicos y útiles para que aprenda el algoritmo de machine learning.

Es importante tener en cuenta que estas características sí que suelen ser en muchos casos definidas a mano a partir de los datos en bruto que se quieren utilizar. Esto es una diferencia importante respecto al deep learning, que como veremos más adelante, se suele prescindir de la creación de características a mano. Un ejemplo es el caso de querer utilizar un algoritmo de machine learning para realizar cálculos referentes a la banca. En esta situación lo más posible es que hubiera que definir índices económicos, a partir de los datos que se tienen, para obtener un buen resultado.

Continuando con el flujo de un sistema de machine learning, una vez tenemos los vectores con las características con las que vamos a entrenar el sistema, junto a estos se deben de dar los resultados esperados para cada uno de los casos (indicado en la figura 3 como "labels"). Una vez se tienen todos estos datos, el sistema empieza un proceso de entrenamiento, donde busca aprender a partir de las entrada dadas, para así conseguir un modelo predictivo eficaz.

Una vez que hemos entrenado a este modelo, se puede utilizar este resultado para obtener una predicción a partir de nuevos datos. Es decir, una vez llegado a este punto se pueden utilizar datos de los que no se conoce el resultado para extraer de nuevo el vector de características, y a partir de este y el modelo predictivo, obtener un resultado.

En la figura 4, se muestra un ejemplo práctico de machine learning. Este consiste en el uso de la regresión lineal para calcular la relación entre una variable dependiente (**y**) y una variable independiente (**x**). Con ello se intenta obtener la función que define una recta a partir de los puntos dados.

Para entender mejor los datos que se muestran hay que indicar que a esta función se le ha aplicado un ruido, dentro de un rango, en los puntos obtenidos, para que así haya una mayor dispersión en los valores. Si el algoritmo funciona correctamente, el resultado final debe ser la función de la recta original ignorando en parte el ruido, o al menos una función bastante cercana a la original.

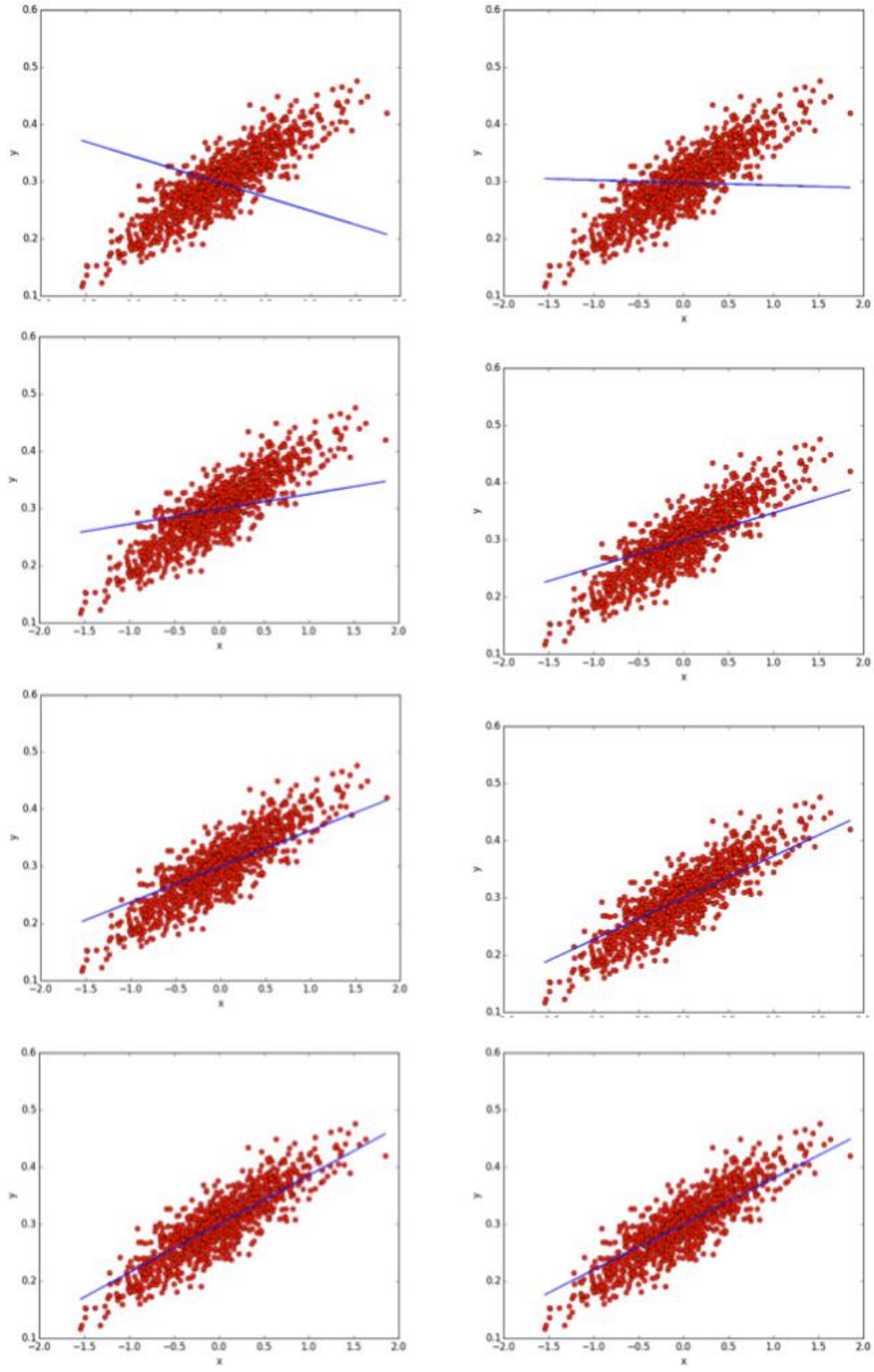


Figura 4. Evolución del resultado de la regresión lineal en las diferentes iteraciones [2].

El resultado de la regresión lineal sería y’ y se puede definir con la siguiente fórmula:

$$y' = w * x + b$$

Fórmula 1. Regresión lineal.

Siendo el valor de **w** un parámetro que se actualiza en cada iteración y **b** un sesgo que también se ajusta a cada iteración.

Lo que se buscaría en el sistema de machine learning sería optimizar los valores **w** y **b,** de la función anterior, para que el resultado se ajuste lo máximo posible al correcto. Pero hay que tener en cuenta que para esto entra en juego otros factores como el ratio de aprendizaje (learning rate), la función de perdida (loss function), el sobreentrenamiento, etc. Pero estos elementos se explicarán más adelante, aplicados al deep learning. Limitándose la información que se acaba de dar a ser un medio para ilustrar un ejemplo de machine learning a modo introductorio.

2.2. Introducción al deep learning

Una vez que te tiene un conocimiento base de lo que es el machine learning, se puede describir el deep learning como una profundización de esta idea. Al igual que el machine learning, el deep learning buscar crear un sistema capaz de aprender y optimizarse a partir de un conjunto de datos para obtener un resultado lo más correcto posible, pero bajo una visión de un sistema de capas conectadas entre sí, las cuales van profundizando y compartiendo la información entre ellas.

La unidad básica en la que se basan las redes neuronales, que conforman el deep learning, es el perceptrón o neurona. Estas intentan ser una adaptación de las neuronas humanas y dan muestra de inspiración biológica en los inicios del machine learning (McCulloch and Pitts, 1943) [3]. Además, aparte del perceptrón en sí, la forma en la que se enlazan también tiene inspiración en el cerebro humano, creando una red de perceptrones.

A pesar de que los perceptrones y las neuronas son conceptos parecidos en el campo del machine learning, tienen ciertas diferencias. Por una parte, el perceptrón es el predecesor de la neurona, siendo este primero asociado a una regresión lineal seguida de una función de salida de tipo salto. Esta función se basa en un umbral que hace que devuelva un valor de 0 o 1. Mientras que en el caso de la neurona, la función de salida puede ser más compleja, pudiendo dar otros valores que no son ni 0 ni 1, a diferencia de los perceptrones.

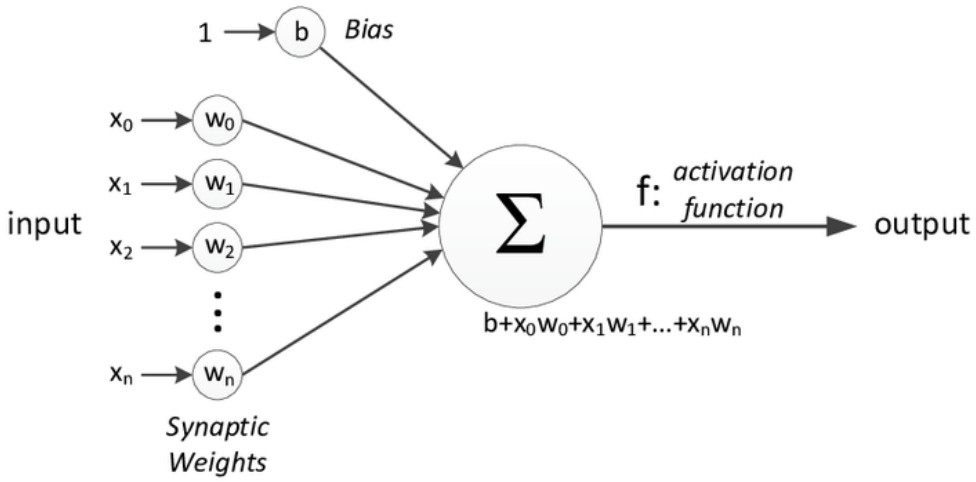


Figura 5. Esquema de una neurona [4]

La fórmula que representa la parte de la regresión lineal de la neurona es la siguiente:

$$y' = \sum_i w_i x_i + b$$

Fórmula 2. Regresión lineal de una neurona

Siendo los valores de **w** parámetros que se ajustan a cada iteración y **b** un sesgo que también se ajusta a cada iteración.

Como se puede ver en la figura 5, la neurona realiza una regresión lineal, que varía respecto a la mostrada anteriormente en que esta tiene diferentes entradas **x**, por lo que pasa a ser un sumatorio y a tener también un vector de valores **w**, uno por cada valor **x** de entrada. Además, este tiene una función de activación **f**. Dos funciones de activación comunes en deep learning son la función **sigmoide** o **ReLU**, cuyo objetivo es aplicar transformaciones no lineales a los valores recibidos. Estas funciones se explican más adelante en este trabajo.

También es importante definir que en el deep learning las neuronas se agrupan en redes neuronales, siendo estas una secuencia de capas compuestas por las neuronas. La figura 6 muestra el ejemplo de una red neuronal basada en capas totalmente conectadas, siendo estas un tipo de capa donde todas las neuronas tienen como entrada todas las que se encuentran en la capa anterior. Finalmente, la red acaba en una última neurona que será la encargada de dar el valor de salida final.

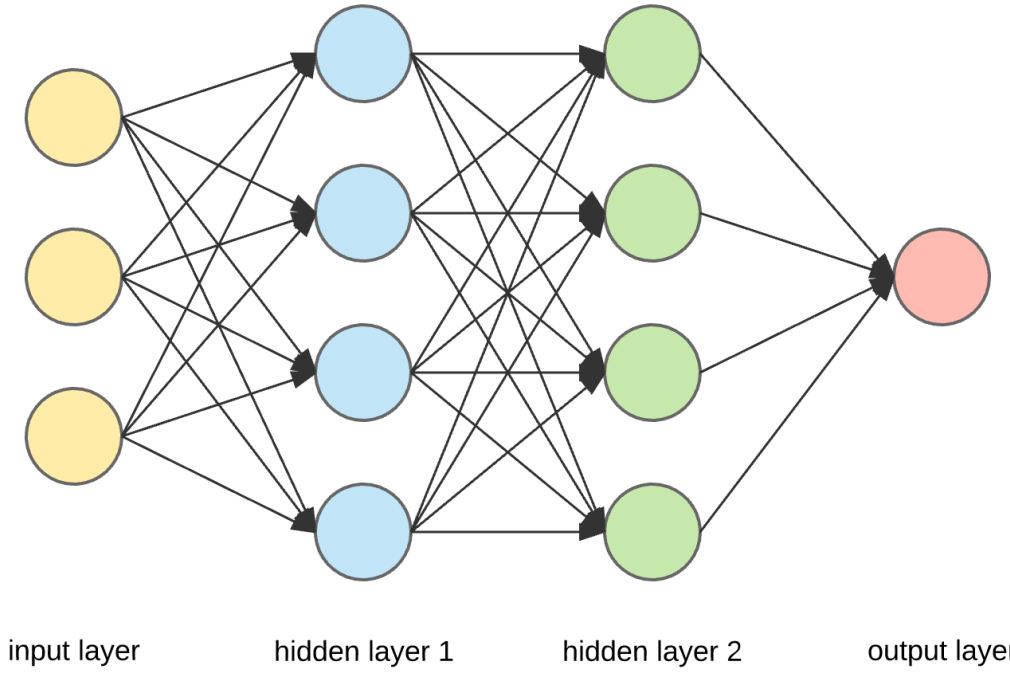


Figura 6. Representación de una red neural basada en capas totalmente conectadas [9].

Respecto a la nomenclatura usada para nombrar las capas de un red neuronal suele estar compuesta por los siguiente elementos:

- Capa de entrada o input layer: Es la primera capa, siendo la entrada a la red. Suelen ser los propios valores de entrada y a la hora de contar el número de capas no suele tenerse en cuenta. En la figura 6 corresponde a la capa de las neuronas de color amarillo.
- Capas ocultas o hidden layers: Son las capas que no se suele conocer su valor de salida, en la figura 6 corresponden a las neuronas de color azul y verde.
- Capa de salida u output layer: Es la capa que contiene los neuronas que dan los valores resultantes buscados. En la figura 6 corresponde a la neurona de color rosado.

Respecto al número de capas de la red usada de ejemplo, en base a la convención más común, tendría 3 capas. Ya que como se ha explicado anteriormente, la capa de entrada no se tiene en cuenta a la hora de contar el número de capas.

En este estudio vamos a profundizar en las CNN. Las CNNs son un tipo de red neuronal, dentro del deep learning, enfocada a su uso en imágenes. Estas utilizan convoluciones, junto a otro tipo de elementos complementarios, para obtener su salida.

Dentro de las CNNs hay diferentes estructuras y variaciones, bien por la evolución que han sufrido o por el objetivo que buscan cumplir. Existe un grupo de redes enfocadas en clasificar a la imagen como un conjunto, por ejemplo, si la foto es de un perro, un gato o ninguno. Otro se encarga de indicar los diferentes elementos dentro de una misma imagen, en este caso se indicaría el área donde se encuentra cada perro o gato. También existe un grupo donde, en vez de mostrar el área, se indicaría a nivel de pixel donde se encuentran estos elementos. En este trabajo se explicarán estos tres grupos de CNNs y cuál de ellos es el más adecuado para nuestro objetivo, así como las bases en común que tienen.

2.3. Elementos de las CNNs

2.3.1. Las CNNs y las convoluciones

Una de las bases de las CNNs son las convoluciones, en concreto las convoluciones discretas. Las convoluciones suponen la multiplicación elemento a elemento de la entrada, denotada por **x**, por los valores de un filtro (o kernel) **w**, siendo este filtro una matriz de dimensiones **m**,**n**. La fórmula matemática que representaría estas convoluciones es la siguiente:

$$z_{i,j} = \sum_m \sum_n x_{i-m,j-n} w_{m,n}$$
$$z, x, w, m, n, i, j \in \mathbb{R}$$

Fórmula 3. Fórmula de una convolución.

El objetivo de las convoluciones es la detección de elementos característicos, por ejemplo, la detección de bordes. Un ejemplo de kernel 3x3 podría ser el siguiente:

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 7. Valores de un filtro de outline.

Estos valores en concreto corresponden a los de un filtro de **outline**, también conocido como filtro de bordes. Este filtro tiene como objetivo el resaltar cuando existe una gran diferencia entre los valores de los píxeles cercanos, mostrando en negro las posiciones donde los pixeles no presentaban una gran diferencia y en blanco donde esa diferencia era mayor. Como ejemplo visual se ha aplicado este filtro sobre una imagen, que corresponde a la figura 8, y después se ha añadido el resultado de aplicar el filtro sobre esta.

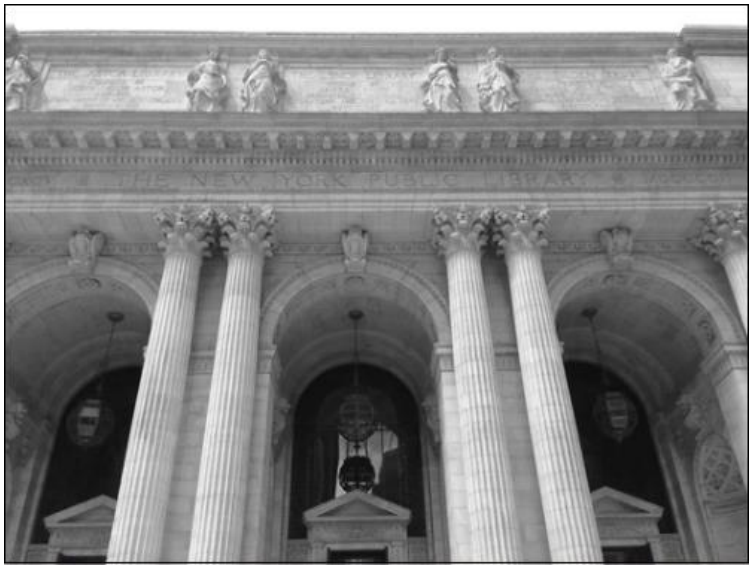


Figura 8. Imagen original sobre la que se aplicará un filtro de bordes [10].

Figura 9. Imagen resultante de aplicar un filtro de bordes [10].

Como se puede ver en la figura 9, en la imagen resultante los bordes de los elementos de la imagen se pueden apreciar fácilmente. Lo que permitiría que en algunos problemas

relacionados con la visión se trabajase mejor con esta imagen que con la original. Siendo estos filtros una de las bases que permiten a las CNNs obtener buenos resultados.

Hasta ahora se ha mostrado la realización de una convolución en imágenes de escala de grises, ya que estas son las que tienen sus valores en una sola capa y son más sencillas de utilizar como ejemplo inicial. Pero hay que tener en cuenta que muchas veces se trabaja con imágenes en RGB, u otro formato que puede tener también varias capas de profundidad como HSV. Por lo que hay que saber que las convoluciones en las CNNs son aplicables tanto a entradas de m x n x 1 como a entradas de m x n x d. Siendo generalmente m x n x 1 en caso de las imágenes en escala de grises y m x n x 3 para imágenes en RGB.

El cambio entre aplicar una convolución a una imagen con una sola capa de profundidad, y otra con tres capas de profundidad consiste en que el kernel pasa a tener también tres dimensiones de profundidad. En estas convoluciones se realizan tres convoluciones por separado, como se realizarían sobre una imagen de escala de grises, pero cada una de las convoluciones sobre su respectiva capa y finalmente se suma el resultado de las tres convoluciones individuales para dar el resultado final.

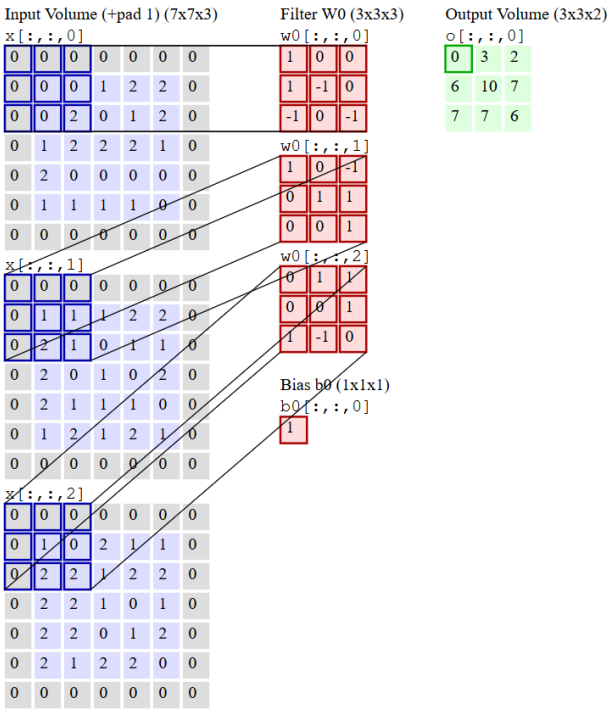


Figura 10. Convolución de una entrada de tamaño 7x7x3 y un filtro de tamaño 3x3 con un padding de 1 y un stride de 2, así como su matriz resultante de tamaño 3x3. [5]

Como se puede intuir viendo la figura 10, en las CNNs las convoluciones tienen parámetros extras, como es el padding y el stride.

El stride es un valor que indica cuantas posiciones en la matriz se recorren entre el cálculo de una convolución en una posición de la matriz y el cálculo en la siguiente. El stride por defecto es 1, y en este caso se realizará el cálculo de la convolución en todas las posiciones de la matriz que se da como entrada. Pero si el stride se establece en 2, como en la figura 10, el cambio se hace realizando saltos de dos posiciones. En sentido horizontal se pasaría de comenzar la operación en la columna 1 a la columna 3, de la 3 a la 5 y así respectivamente. En sentido vertical pasaría lo mismo, pasando de la fila 1 a la 3, de la 3 a la 5 y así respectivamente.

También se puede observar en la figura 10, que se ha añadido un padding de 1 píxel alrededor de la matriz de los valores de entrada con valores 0. Siendo esta otra de las opciones que se pueden establecer referente a las convoluciones. Indicando si se va a añadir o no un relleno a la imagen dada. El padding es útil tanto para evitar reducir el tamaño de la entrada, como para ayudar a que se obtenga información de los bordes de esta. Sin este, en las redes con muchas capas convolucionales, la imagen de entrada podría acabar reducida a dimensiones muy pequeñas.

Como se ha explicado anteriormente, el objetivo de las convoluciones es la detección de características, como los bordes. Pero estas características son las que se detectan en los primeros niveles de la red, siendo características muy sencillas que no permitirían una detección de elementos complejos por sí mismas de forma aislada. Es por eso que al profundizar en los diferentes niveles de las redes neuronales convolucionales las características (o features) detectadas van incrementando su complejidad. Pasando de ser simplemente la detección de bordes o cambios en los valores de un píxel respecto a los que se encuentran próximos a este, a realizar una detección de componentes que se aproximan a elementos distinguibles del objeto a buscar. En la figura 11 se puede encontrar un ejemplo simplificado de esta progresión a través de las capas de las CNNs.

Como se puede observar también en la figura 11, un elemento clave en las CNNs es que cada uno de los kernels generados se centra en buscar un característica diferente. Por lo que al final la red acaba siendo compuesta por un gran conjunto de características que se entrelazan entre sí para detectar los diferentes elementos complejos. Este funcionamiento es el que le da una gran capacidad de detección a las redes neuronales convolucionales.

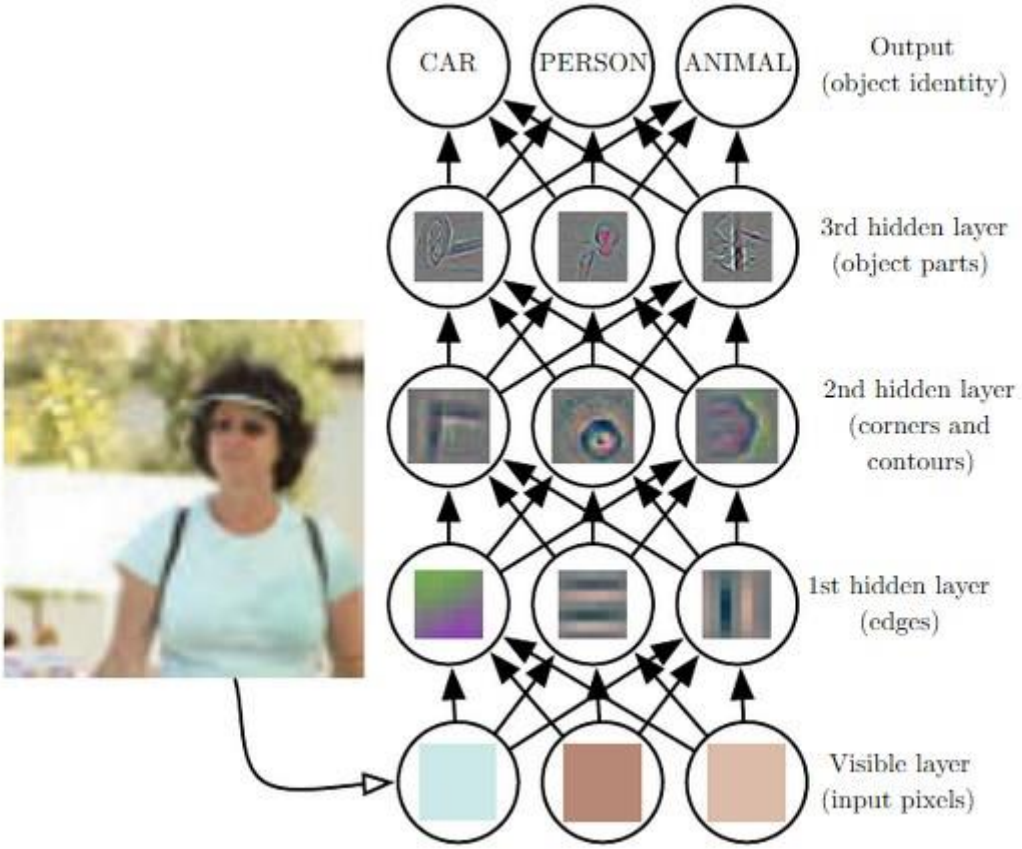


Figura 11. Evolución de la información obtenida por las convoluciones respecto al número de capas [1].

Como muestra para mejor ilustración de la idea de detección de características complejas, por parte de las convoluciones, se ha añadido la figura 12.

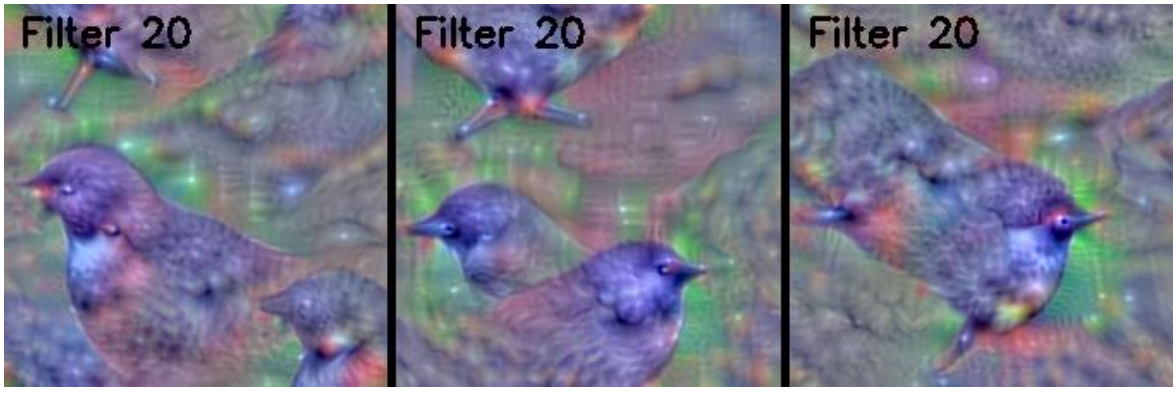


Figura 12. Evolución de las formas/texturas buscadas por un filtro, en este caso el filtro número 20 de una red, en tres iteraciones diferentes durante el entrenamiento de una CNN [11].

En la figura 12 se puede ver para qué patrones estaba siendo entrenado el filtro de convolución de una determinada capa. Como se puede intuir por el resultado obtenido, la red neuronal convolucional a la que pertenece este filtro estaba siendo entrenada para buscar pájaros. Sirviendo esta caso como ejemplo para mostrar de forma visual la detección de características más complejas.

2.3.2. Función de activación

Las funciones de activación son las encargadas de aplicar transformaciones no lineales a valores obtenidos en la red. La utilización de estas funciones permite a las redes neuronales generalizar mejor y adaptarse a una mayor variedad de datos. Definiendo más el problema, estas funciones de activación permiten añadir un elemento de no linealidad [13], sin el cual el resultado de las redes se limitaría a una función lineal, disminuyendo su capacidad de predicción. Esto último es debido a que los resultados de salida serían siempre una función lineal, por lo que simplemente se limitaría a ser operaciones lineales de las entradas.

Sigmoid

La función sigmoid devuelve valores que se encuentran entre 0 y 1. Los valores de salida de la función varían en gran medida cuando los valores de entrada son cercanos a 0, pero cuando los valores de entrada se encuentran alejados de 0 el cambio entre valores de salida suele ser pequeños. La fórmula que define esta función es la siguiente:

$$a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$a, z \in \mathbb{R}$$

Fórmula 4. Función sigmoid

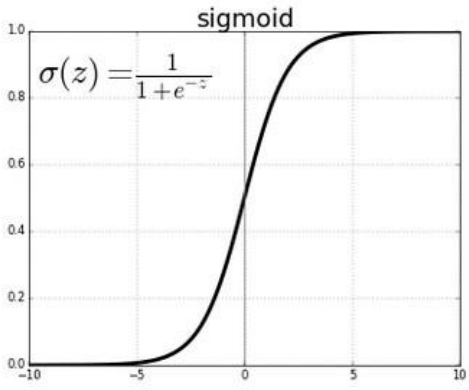


Figura 13. Representación de la función sigmoid [12].

Actualmente esta función no suele ser muy utilizada como función de activación, ya que para la salida de las convoluciones se suele utilizar la función ReLU o Leaky ReLU, en menor medida esta última.

Tanh

La función tanh devuelve un valor entre -1 y 1. Al igual que con la función sigmoid, los valores de salida de la función varían en gran medida cuando los valores de entrada son cercanos a 0. Pero cuando los valores de entrada se encuentran alejados de 0 el cambio entre valores de salida suele ser pequeños. La fórmula que define la función tanh es la siguiente:

$$a = tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$a, z \in \mathbb{R}$$

Fórmula 5. Función tanh

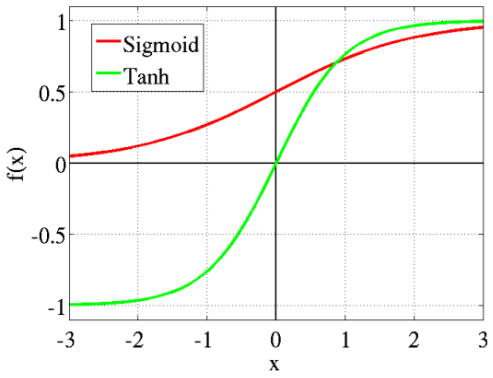


Figura 14. Representación de las funciones Sigmoid, en rojo, y Tanh, en verde [12].

Esta función de activación se suele utilizar en las salidas de las capas que no son la capa final. Pero no suele ser muy utilizada, ya que en su lugar se suele utilizar la función ReLU, o en menor medida, Leaky ReLU.

ReLU

La función ReLU devuelve el valor 0 o **z**, siendo **z** el valor de entrada a la función. Su funcionamiento se basa en que si la entrada es menor a 0 la función devuelve 0 y sino el valor de entrada a la función. La fórmula que define a esta función es:

$$a = R(z) = max(0, z)$$

$$a, z \in \mathbb{R}$$

Fórmula 6. Función ReLU

Esta función de activación es la que más se utiliza en las capas que no son la capa de salida, ya que en general suele dar mejores resultados que sigmoid y tanh.

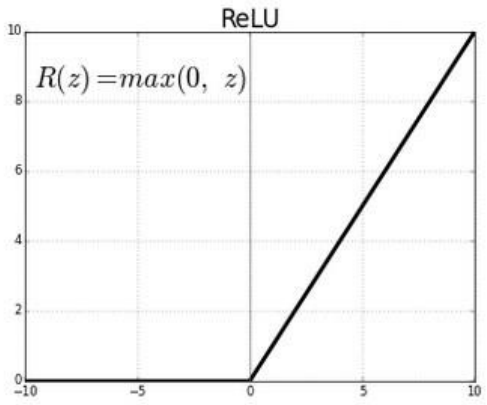


Figura 15. Representación de la función ReLu [12].

Leaky ReLU

La función Leaky ReLU es una variante de ReLU. En este caso la función devuelve un valor de **s*z** o **z**, siendo **z** el valor de entrada a la función y **s** un valor por el que se multiplica el valor de entrada en caso de que su valor sea menor que 0. La variable **s** suele tener un valor pequeño, ya que su función es reducir el valor de **z** para que cuando **z** sea menor que 0 el cambio de valor en la salida sea menor a cuando es mayor a 0. De esta manera se consigue una transformación no lineal y que el valor obtenido no sea 0 cuando la suma ponderada de la entrada sea menor a 0, pudiendo así las neuronas seguir modificando sus pesos. La fórmula que define a esta función es:

$$a = R(z) = max(s * z, z)$$

$$a, z, s \in \mathbb{R}$$

Fórmula 7. Función Leaky ReLU

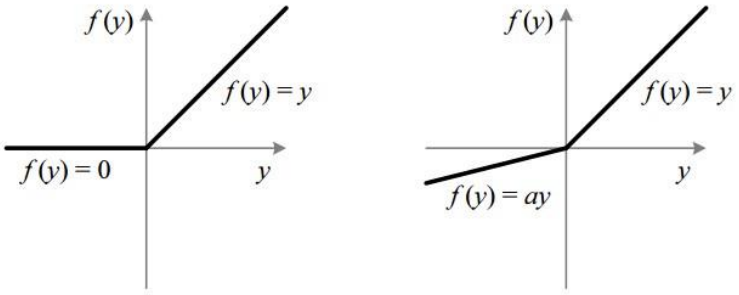


Figura 16. Representación de las funciones ReLU y Leaky ReLU respectivamente [12].

Esta función suele dar resultados muy parecidos a los obtenidos con ReLU, llegando a mejorar su resultado en ocasiones [14]. Pero a pesar de ello está más extendido el uso de ReLU que el de Leaky ReLU.

2.3.3. Loss

El **loss function**, o función de perdida en español, es una función que se encarga de calcular como de mal se ha realizado una predicción en base a los datos dados y el resultado obtenido. Para calcular este error obtenido hay diferentes funciones, que se deben utilizar dependiendo del problema que se quiera afrontar.

La principal clasificación que podemos hacer de estas funciones es entre aquellas se usan para calcular una regresión (**regression losses**) y una clasificación (**classification losses**). Las de regresión se utilizan cuando se quiere obtener la predicción de un valor continuo, por ejemplo, el precio de un inmueble. En cambio, las de clasificación se utiliza cuando se quiere clasificar elementos en un conjunto finito de elementos. A continuación, se van a describir algunas funciones de ambos tipos como ejemplo.

Regression losses

Mean square error / Quadratic loss / L2 loss

El mean square error, o el error cuadrático medio en español, sirve para estimar el promedio de los errores al cuadrado. Este error solo mide la cantidad de error y no el signo de este, debido a que eleva al cuadrado. Una característica de esta función es que penaliza mucho lo errores mayores, también debido al uso del elevado al cuadrado. La fórmula de este error sería:

$$MSE = \frac{\sum_i^n (y_i - y'_i)^2}{n}$$

Fórmula 8. Mean Square error

Siendo **n** el número de casos mientras que **y** e **y'** el resultado esperado y el obtenido respectivamente.

Mean absolute error / L1 loss

El mean absolute error, o error absoluto medio en español, sirve para estimar el promedio de los errores usando el valor absoluto. También es independiente del signo del error, al igual que el MSE. Este cálculo del error penaliza menos los grandes errores, pero es más robusto ante los errores atípicos. La fórmula de este error sería:

$$MAE = \frac{\sum_i^n |y_i - y'_i|}{n}$$

Fórmula 9. Mean absolute error

Siendo **n** el número de casos mientras que **y** e **y'** el resultado esperado y el obtenido respectivamente.

Classification losses

Hinge loss / Multi class SVM loss

Se basa en que el resultado de la categoría correcta debe ser mayor que la suma de todas las categorías incorrectas, teniendo al menos un margen de diferencia mínimo por seguridad. La fórmula por la que se rige es:

$$SVM\ Loss\ = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

Fórmula 10. SVM Loss

Donde **j** es el subíndice que indica la posición del resultado correcto, e $\boldsymbol{y_i}$ indica los valores del resto de casos.

Cuanto menor sea el loss obtenido, mayor será la probabilidad que la predicción haya sido correcta. Por ello se busca minimizar el loss obtenido en las diferentes clases.

Cross Entropy Loss

Esta es una de las funciones de pérdida más utilizada para la clasificación. Tiene como ventaja que penaliza fuertemente cuando tiene una predicción con una gran probabilidad, pero resulta ser incorrecta. La fórmula de esta función sería:

$$Cross\ Entropy\ Loss\ = -(y_i \log(y'_i)\ +\ (1\ -\ y_i) * \log(1 - y'_i))$$

Fórmula 11. Cross Entropy Loss

Donde **y'** es el resultado obtenido, **y** el resultado esperado e **i** indica el caso dentro del conjunto de datos obtenido o dados.

El uso del Cross Entropy Loss va estrechamente relacionado con la función softmax, ya que los valores **y'** mostrados en la fórmula anterior se obtienen tras aplicar esta. El funcionamiento de softmax se basa en que dado un conjunto de **x** números se obtiene una distribución de valores que da como suma de estos el valor 1, indicando cada uno de estos componentes la probabilidad de una clase diferente. Por ejemplo, dada la entrada [2.0 1.0 0.1] el resultado sería [0.7 0.2 0.1], representando estos tres últimos números tres probabilidades.

La función softmax se define con la siguiente fórmula:

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Fórmula 12. Softmax

Siendo **y** el valor dado, **j** el subíndice de los casos diferentes al dado, e **i** el subíndice del caso dado.

2.3.4. Gradient descent

El **Gradient Descent** (descenso por gradiente en español) es un algoritmo que se utiliza para entrenar el modelo. Es un algoritmo de optimización que ajusta los pesos de la red de forma iterativa para minimizar una función hacia el mínimo global, aunque puede darse la situación de que lo que encuentre sea un mínimo local. Simplificando, su objetivo es encontrar aquellos valores que minimizan el error de la función de coste.

El descenso por gradiente se inicializa con unos valores aleatorios que se van ajustando iteración tras iteración. Para definir cuanto se ajustan los valores en cada iteración existe un hiperparámetro llamada **learning rate**, el cual indica el factor de corrección por cada iteración. Por ejemplo, si el learning rate (ratio de aprendizaje en español) se establece en 0.0001, en cada iteración se aplicará una corrección equivalente al gradiente del error con respecto a cada uno de los parámetros del modelo multiplicado por 0.0001.

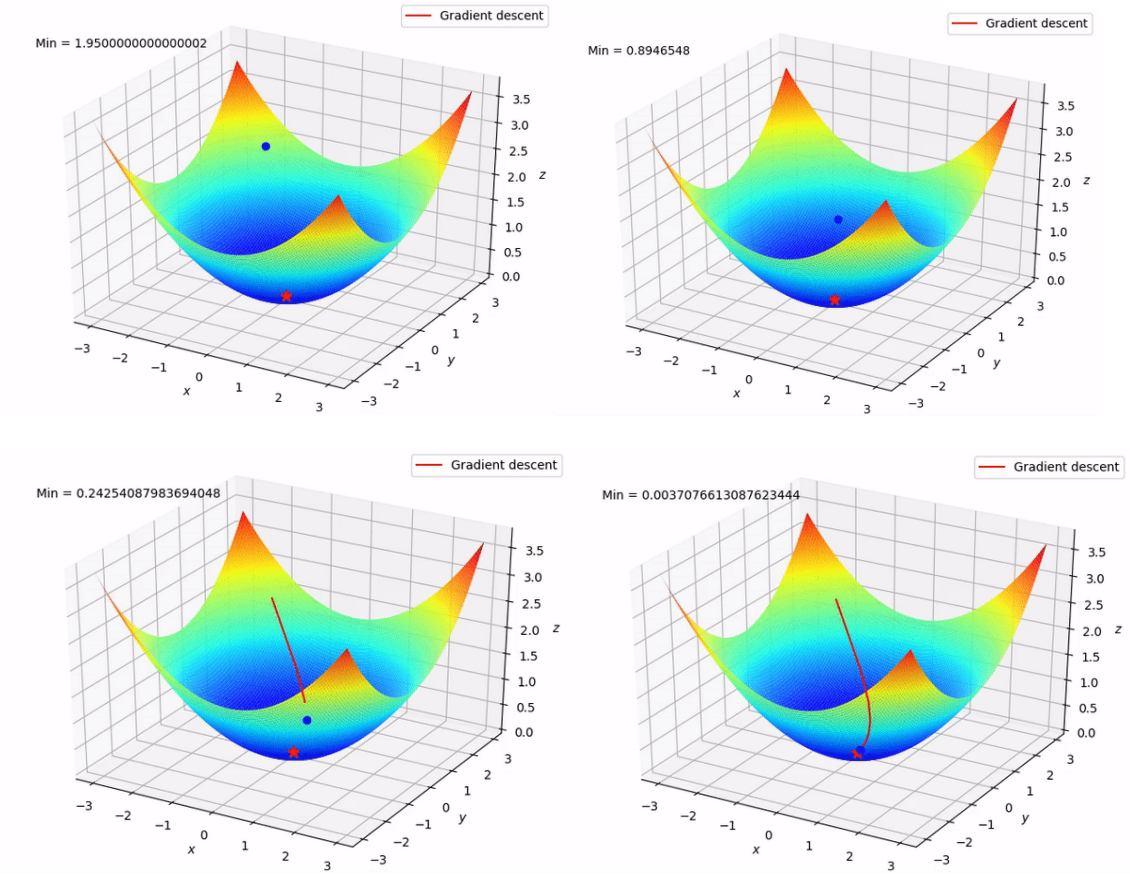


Figura 17. Representación gráfica del descenso por gradiente a través de diferentes iteraciones [15].

La figura 17 ilustra de forma gráfica cómo evoluciona el error a lo largo de las diferentes iteraciones del descenso por gradiente. Pasando desde un error de 1.95 a uno cercano a 0.0037. Como se puede ver, y se ha explicado anteriormente, este proceso consiste en reducir el error iteración a iteración, en este caso siendo el error la diferencia entre la posición objetivo, señalada en rojo, y la posición actual, señalada en azul.

Llegado a este punto es importante señalar la diferencia entre hiperparámetros y parámetros. Siendo los hiperparámetros los valores que se definen a priori y son constantes durante todo el proceso de entrenamiento y uso de la red. En cambio, al hablar de parámetros nos referimos a aquellos valores que se van modificando durante el entrenamiento del modelo. Por ejemplo, el número de entradas de una red o el learning rate son hiperparámetros, mientras que los valores de los filtros de las convoluciones son parámetros, ya que van cambiando a lo largo del proceso de entrenamiento de la red.

Forward Propagation

Esta es la etapa en la que se realiza una propagación hacia delante. Es decir, es cuando se calcula la salida de la red neuronal a partir de una entrada dada. En esta etapa se busca obtener un resultado y no se calcula un error para realizar una rectificación, como pasa durante el momento del **backpropagation**.

Backpropagation

Esta es la etapa en la que se realiza una propagación hacia atrás. Es cuando se calcula los errores respecto a la salida esperada y se aplica una corrección de los valores de la red neuronal para minimizar este error. Para establecer la cantidad de corrección se establece un parámetro llamado **ratio de aprendizaje** (en inglés **learning rate**).

Este momento es cuando la red realmente "aprende", es decir, cuando sus parámetros se ajustan buscando acotar una función que defina lo mejor posible el problema a solucionar. La fórmula que representa la actualización de los pesos de la red sería:

$$w = w - \alpha * \frac{dL}{dw}$$

Fórmula 13. Backpropagation

Siendo **w** el peso sobre el que se calcula el error, $\boldsymbol{\alpha}$ el learning rate y $\frac{\boldsymbol{dL}}{\boldsymbol{dw}}$ la derivada del error respecto al peso, por lo que se podría interpretar como la forma en la que varía el error de la salida en función del valor del parámetro **w**.

Batch Gradient Descent

El descenso por gradiente tiene diferentes implementaciones en su uso práctico, la primera de ellas es el batch (o "standard") gradient descent. Esta se podría considerar la implementación base del descenso por gradiente.

Su implementación consiste en el cálculo del error total sobre el conjunto de ejemplos y la corrección de este error en cada iteración durante el entrenamiento de la red. Es decir, a cada iteración del algoritmo se obtiene el error medio a partir del conjunto de casos y este es el valor que se utiliza para realizar una corrección de los pesos de la red.

El gran problema que nos encontramos con esta implementación del descenso por gradiente es que, al obligarnos a usar todo el conjunto de datos en cada iteración, se da una

convergencia más lenta y tiene el riesgo de quedarse en mínimos locales. Por otra parte, para evitar tener que cargar todos los datos simultáneamente se puede utilizar un acumulador, para que al acabar todos los casos se pueda aplicar la corrección que se ha ido calculando.

SGD

A diferencia del batch gradient descent el SGD (Stochastic gradient descent) no realiza la corrección de los parámetros con el conjunto de datos de prueba entero, sino que elige un ejemplo aleatorio y se optimiza a partir de este y avanza al siguiente ejemplo elegido del conjunto de datos. Es decir, con SGD la red es entrenada sobre un caso, el error es calculado, y los pesos variados en base a este, tras lo que pasa al siguiente. Este proceso se repite hasta terminar el conjunto de datos sobre el que está siendo entrenada la red, dando por concluida una época, pudiendo tener el entrenamiento varias. Este algoritmo tiene varias ventajas frente al batch gradient descent:

- Al no necesitar cargar todos los datos de entrenamiento a cada iteración, ya que no corrige el error a partir de todo el conjunto, la cantidad de memoria requerida por este proceso es menor.
- Suele converger en una solución óptima de forma más rápida que el batch gradient descent.
- Puede salir de mínimos locales de forma más sencilla en busca de mejores soluciones.

Hay que tener en cuenta que SGD, al contrario que cuando se realiza el entrenamiento mediante Batch Gradient Descent, no tiene unos ajustes suaves, sino que los valores tienen unos cambios más bruscos.

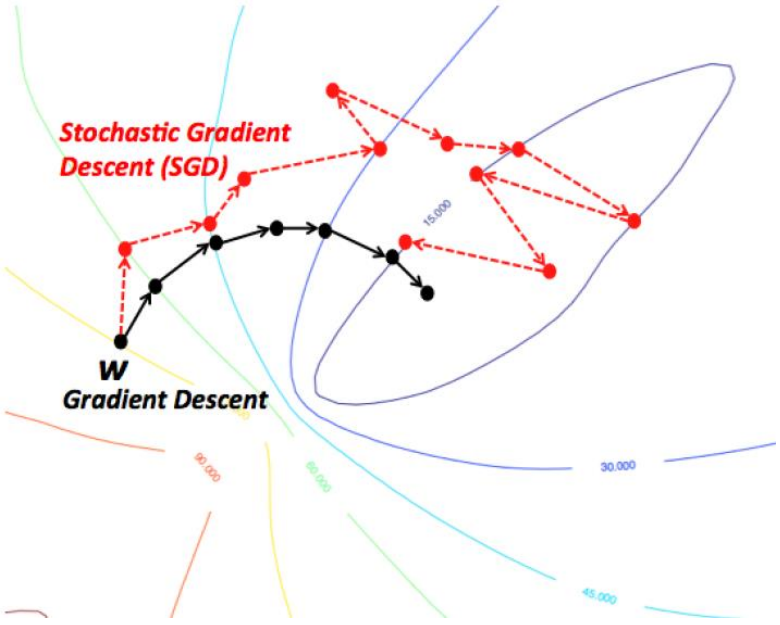


Figura 18. Comparación del proceso de optimización usando Batch Gradient Descent y SGD [16].

Mini-batch gradient descent

Este algoritmo supone un punto medio entre el batch gradient descent y SGD. Este se basa en la separación del conjunto de datos en porciones, llamadas **mini-batchs**. En cada iteración se calcula el error sobre los datos que se encuentran contenidos dentro de este mini-batch, corrigiéndose los pesos en base a este, completándose una época al recorrerse todos ellos.

Frente a SGD tiene la ventaja de que al no ser un único ejemplo cada vez se pueden usar técnicas de paralelización sobre los datos que se están usando, mejorando así el rendimiento durante el entrenamiento. También hay que destacar que cuando se compara mini-batch y SGD en el primer caso los ajustes suelen ser más suaves, por lo que se producen cambios de valores más controlados.

2.3.5. Fully conected layer

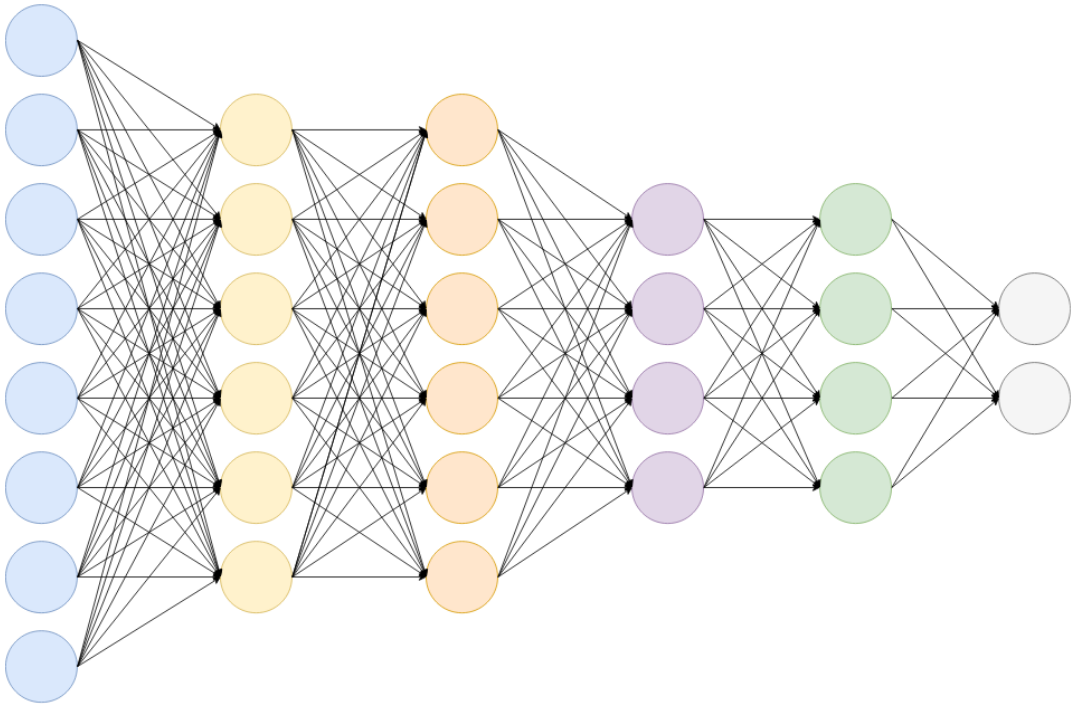


Figura 19. Red neuronal compuesta por capas totalmente conectadas [17].

Las capas totalmente conectadas, o fully conected layers en inglés, consisten en capas donde todas las entradas de las neuronas están conectadas a todas las salidas de la capa anterior. Las redes neuronales más simples se basan en el uso exclusivo de este tipo de capas. En redes más complejas, que hacen uso de capas convolucionales, se suele utilizar para producir la salida final utilizando el vector de características generado en etapas previas.

En el caso de que sea necesario, en las CNNs se realiza un aplanamiento de la salida de la capa anterior para poder utilizarla como entrada de la capa totalmente conectada. Es decir, la salida se convierte en vector unidimensional de tamaño fijo sobre el que pueda trabajar

la siguiente capa. Por ejemplo, si la salida de la capa anterior es de 3x3 se pasa a una salida que es un vector unidimensional de 9 elementos que se podrá utilizar en la capa totalmente conectada.

2.3.6. Pooling layer

Estas capas son utilizadas con el objetivo principal de reducir el número de parámetros en la siguiente capa, esto tiene diversas ventajas:

- Ayuda a reducir el overfitting (sobreentrenamiento en español).
- Reduce el coste computacional de la red en su conjunto.
- Ayuda a extraer mejores elementos característicos a la red, ya que ayuda a la generalización.

Aunque existen diferentes tipos de pooling layers, las que más se suelen utilizar son:

- Max pooling
- Average pooling

Max pooling

El funcionamiento de las pooling layers, que utilizan la técnica de max pooling, se basa en aplicar un filtro de tamaño NxN que selecciona el mayor de los valores de una matriz de NxN posiciones, recorriendo todos los datos dados como entrada, generando una salida en base a los valores que se han ido obteniendo.

Por ejemplo, tenemos una capa de max pooling con un filtro de 2x2. Lo que haría esta capa es que, a partir de los 4 valores que se ven afectados por cada uso del filtro, se seleccionaría el mayor valor de ellos y ese valor sería el valor de salida correspondiente a esa posición respecto a la matriz de entrada. Esto se puede ver más claramente en la figura 20, ya que esta es un caso de pooling layer que usa max pooling.

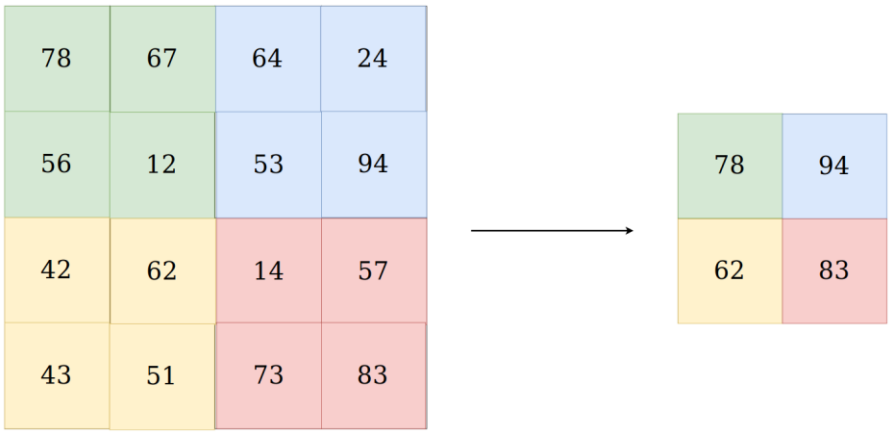


Figura 20. Ejemplo de max pooling con stride de 2 y kernel de 2x2 [17].

Este proceso, como se ha dicho anteriormente, ayuda a que consiga una mayor generalización. Esto es debido a que ayuda a eliminar algunos efectos de la imagen, como

por ejemplo la distorsión, ya que permite mejor generalizar los datos. Este tipo de capa de pooling suele ser la más utilizada.

Average pooling

Este es el segundo tipo de pooling layer más utilizado, llamado average pooling. Este se diferencia del caso de max pooling únicamente en el proceso con el que se obtiene el valor de salida.

En el caso anterior, a partir de los valores seleccionados se obtenía el mayor de los valores y este era utilizado como parte de la salida, pero con average pooling lo que se realiza es la media de los valores seleccionados, dando esta media como salida. Por ejemplo, si se utiliza un filtro de 2x2, siendo las entradas 10, 10, 20 y 20, el valor de salida será de 15.

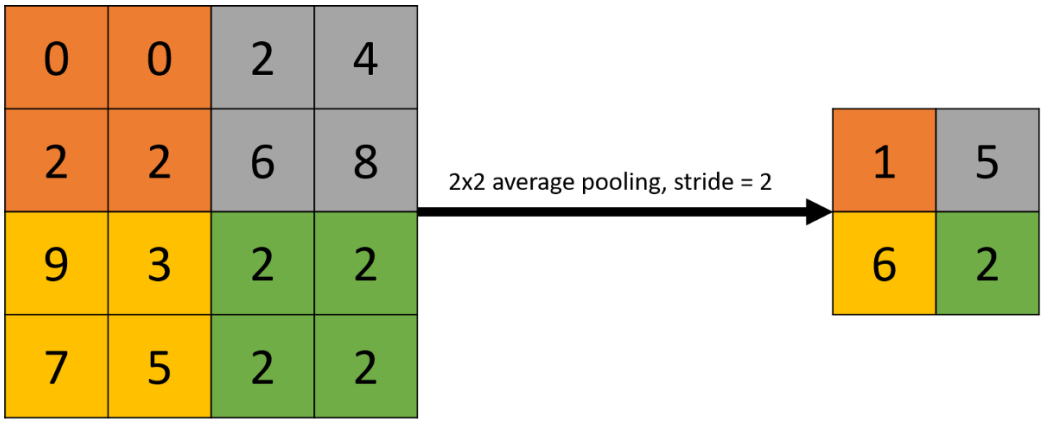


Figura 21. Ejemplo de average pooling con stride de 2 y kernel de 2x2 [25].

2.3.7. Regularización

La regularización es una técnica que es utilizada para intentar reducir el sobreentrenamiento, es decir, la obtención de modelos que se ajustan demasiado a los datos dados.

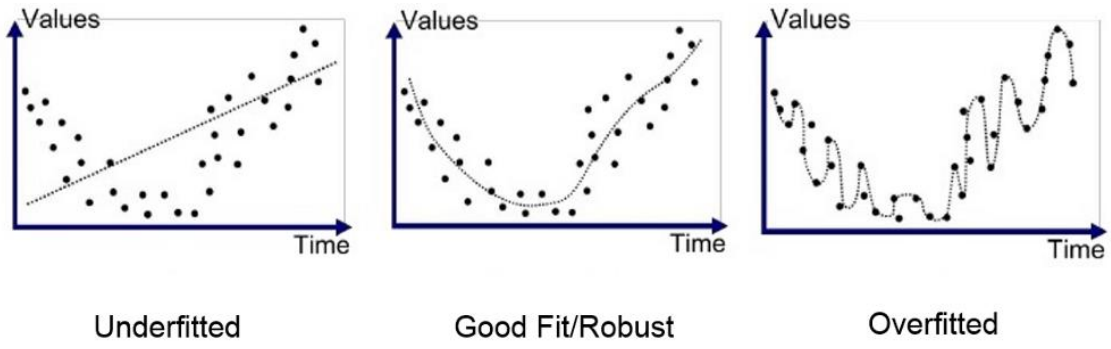


Figura 22. Ejemplo de la predicción realizada por un modelo sin entrenar, bien entrenado y sobreentrenado [19].

La figura 22 sirve como ejemplo para ilustrar el problema del sobreentrenamiento. Cuando el modelo se ajusta demasiado a los datos puede acertarlos durante el entrenamiento, pero sobre el conjunto de datos de test fallarán más, por este sobreajuste.

El estado de una red neuronal en relación al resultado del entrenamiento que se ha realizado se puede separar en tres grupos. El primero es underfitting, que tiene lugar cuando el modelo no se ha entrenado lo suficiente y es incapaz de realizar una buena predicción que se ajuste bien tanto a los datos de entrenamiento como de validación y test. El segundo es good fit, o buen entrenamiento, en el que la red es capaz de realizar una buena predicción en base a los datos.

El último es overfitting, o sobreentrenamiento. En este caso el modelo se ha ajustado demasiado a los datos de entrenamiento, realizando una predicción que no realiza una buena generalización, y que sobre los datos de prueba fallará. Esto provoca que la red en vez de aprender la función que describe los datos de entrada aprenda el ruido de estos, dando lugar a que la predicción se base en este, solo siendo capaz de realizar buenas predicciones sobre los casos en los que la red ha sido entrenada y no sobre aquellos que no ha visto.

El objetivo, en este apartado, es conseguir que la red tenga un buen entrenamiento, evitando tanto el underfitting, como el overfitting. Para intentar minimizar este problema se pueden utilizar técnicas de regularización. A continuación, se muestran tres de las técnicas más utilizadas en deep learning para este propósito.

L1 regularization

La utilización de la regularización L1, L1 regularization en inglés, se basa en añadir al resultado del cálculo del loss un valor extra que ayude en su regularización. La fórmula para calcular este valor se muestra a continuación:

$$L1\ \text{regularization} = \lambda * \sum_{i=1}^{n} |w_i|$$

Fórmula 14. L1 regularization

Siendo $\boldsymbol{\lambda}$ un hiperparámetro, que se utiliza a modo de coeficiente de regulación, que es elegido antes de comenzar el entrenamiento del modelo y **w** los diferentes pesos de la red.

El resultado de combinar la función de perdida, en este caso MSE, y L1 regularization, es el siguiente:

$$MSE + L1\ \text{regularization} = \frac{\sum_i^n (y_i - y'_i)^2}{n} + \lambda \sum_{i=1}^{n} |w_i|$$

Fórmula 15. MSE + L1 regularization

Siendo $\boldsymbol{\lambda}$ un hiperparámetro, que se utiliza a modo de coeficiente de regulación, que es elegido antes de comenzar el entrenamiento del modelo, y **w** los diferentes pesos de la red.

El funcionamiento de esta técnica se basa en que si el valor de un peso es muy grande este ampliará mucho el error calculado mediante la función de perdida. Esto hará que se tienda a predicciones menos adaptadas a casos concretos. Esta técnica suele funcionara bien, pero se debe tener en cuenta que si el valor de λ es demasiado grande se puede caer en el caso del underfitting, es decir, en que haya falta de entrenamiento.

L2 regularization

$$L2 \text{ regularization } = \frac{\lambda}{2}\sum_{i=1}^{n} w_i^2$$

$$MSE \; + \; L2 \text{ regularization } = \frac{\sum_i^n (y_i - y'_i)^2}{n} \; + \; \lambda \sum_{i=1}^{n} w_i^2$$

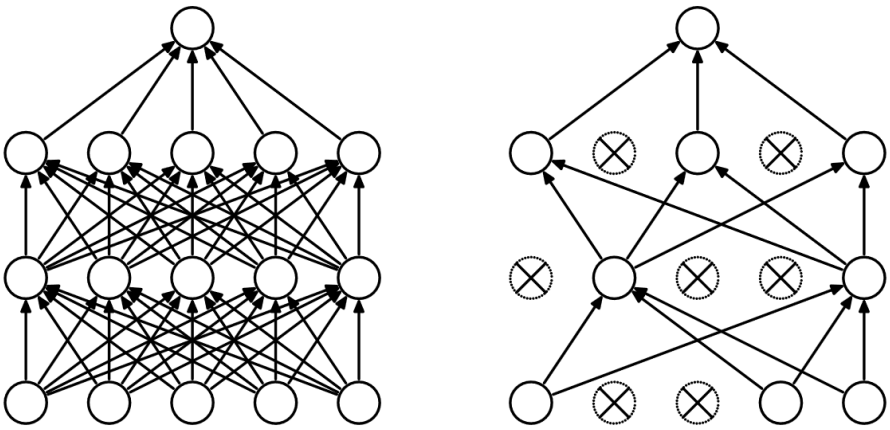
Fórmula 16 y 17. L2 regularization y MSE + L2 regularization, respectivamente.

El funcionamiento de L2 es parecido a L1, con la diferencia que se usa el valor al cuadrado de los pesos. En general L2 suele dar mejores resultados que L1, y por ello suele ser más usado.

“In all cases, logistic regression with L1 regularization, as predicted by the theoretical results, exhibits a significantly higher tolerance to the presence of many irrelevant features.” (Y.Ng, 2004) [20]

Dropout

La técnica del dropout es una técnica relativamente nueva, ya que se planteó en 2014 [18]. Esta consiste en que a cada iteración se descarte la utilización de algunas de las neuronas que componen la red.



(a) Standard Neural Net (b) After applying dropout.

Figura 23. Ejemplo del uso de dropout [18].

Por ejemplo, si se establece un dropout de un 40% en una capa de la red, esto quiere decir que se utilizará, durante la fase de forward propagation y backpropagation, el 60% de la neuronas y el otro 40% no. A cada iteración la decisión de las que son utilizadas y las que no varía.

Los principales efectos que tiene el uso del dropout en las redes neuronales son los siguientes:

- Ayudan a combatir el overfitting, ya que fuerza a la red a utilizar diferentes conexiones entre neuronas, aprendiendo así características más robustas, las cuales relacionan a diferentes conjuntos de neuronas.
- Aumenta el número de iteraciones necesarias para entrenar la red, pero estas iteraciones tienen una duración menor.

2.4. Tipos de CNNs

En lo referente a las CNNs hay tres tipos principales de redes neuronales para realizar detecciones en imágenes. Las redes que realizan una predicción sobre el conjunto de la imagen, las que realizan una predicción por elementos en la imagen y las que realizan una predicción basada en una segmentación semántica a nivel de píxel.

Ya que cada uno de estos grupos tiene diferentes métodos para conseguir su objetivo, en cada apartado se explicará de forma general sus principales características y se mostrará un ejemplo de estos.

2.4.1. Predicción por imágenes

Este primer grupo es el más básico y el primero que se empezó a utilizar. Este tipo de redes neuronales se encargan de clasificar una imagen en categorías, por ejemplo, pueden clasificar imágenes en dos grupos, perro y no perro.

Pero también pueden ser un grupo de etiquetas mucho más amplias, con cientos, o incluso miles, de etiquetas. Este tipo de redes conciben una imagen como un todo, no por elementos divisible dentro de la imagen.

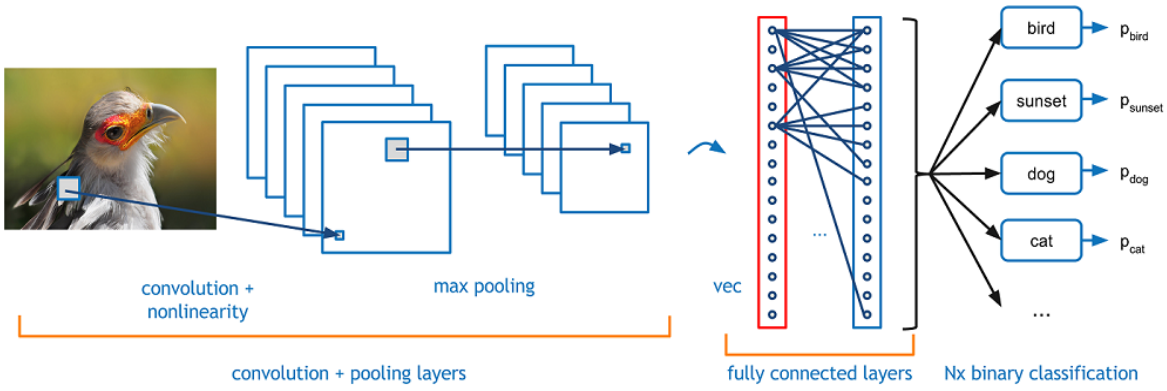


Figura 24. Ejemplo de detección y clasificación de imágenes en categorías mediante CNNs [21].

2.4.2. Predicción por bounding box

Este segundo grupo de CNNs son más complejas y dan más versatilidad que las anteriores. Este tipo de redes ya no etiquetan a una imagen como un conjunto, sino que son capaces de seleccionar los diferentes elementos dentro de estas.

Por ejemplo, si tenemos una imagen donde aparecen un perro y un gato, serían capaces de seleccionar, dentro de un área rectangular, donde esta cada uno de estos dos elementos en la imagen. Esta área es llamada bounding box.

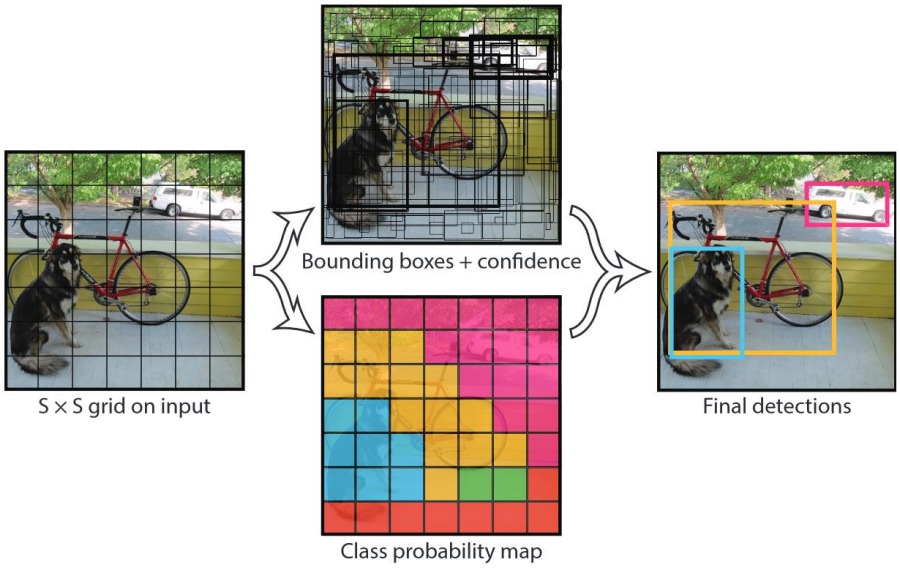


Figura 25. Proceso de detección de objetos en Yolo, una CNN de clasificación por bounding boxes [22].

Suponen una gran ventaja frente al anterior tipo de CNNs, ya que una imagen pasa de ser capaz de ser clasificada en un grupo, a poder a analizar la imagen elemento a elemento. Pudiendo tener resultados de diferentes categorías.

Hay diferentes modelos para este tipo de redes, pero en la figura 25 se puede observar el procedimiento utilizado por Yolo, que se basa en obtener diferentes bounding boxes, que son las áreas rectangulares de las posibles selecciones. Después se seleccionan finalmente las que tienen más probabilidades de ser bounding boxes correspondientes a selecciones correctas.

Si bien el proceso puede cambiar dependiendo de la red utilizada, la obtención del resultado final mediante bounding boxes es un elemento común en este tipo de redes.

2.4.3. Predicción por segmentación semántica.

Este grupo de CNNs son las que realizan una segmentación semántica, es decir, clasifican píxel a píxel la pertenencia de un elemento en un categoría concreta. Siguiendo con el ejemplo del perro y el gato, si en el grupo anterior se detectaba la posición del estos elementos dentro de un área rectangular, en este caso directamente se detectaría en que pixeles se encuentra el elemento a buscar.

Este tipo de red tiene la ventaja, frente a la redes de detección por bounding boxes, de que la precisión de la predicción no se encuentra limitada a un área donde se encuentra un elemento. Ya que, en el anterior tipo, parte del área donde se encuentra predicción puede ser el fondo sobre el que se encuentra el elemento a buscar. En cambio, con este tipo de predicción podemos saber exactamente a nivel de píxel donde se encuentra el elemento que nos interesa.

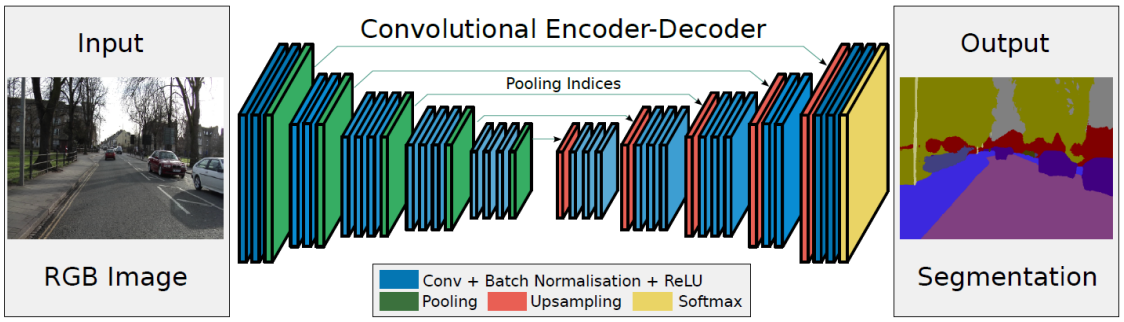


Figura 26. Proceso de detección mediante segmentación semántica en SegNet [23].

A pesar de sus ventajas, estas redes tienen algunos inconvenientes comparadas con las redes que utilizan una detección basada en bounding boxes:

- La precisión suele ser menor a las redes que hacen una detección limitada al área de los elementos que se quieren buscar.
- La cantidad de datasets disponibles de forma pública es menor y, además, tienen una menor cantidad de imágenes.
- La creación de datasets para este tipo de red requieren una mayor cantidad de trabajo, ya que se necesita una anotación de los datos más detallada.

Una de las bases de estas redes es el uso de una parte de encoder seguida de una parte de decoder en la red. Este es su funcionamiento:

- **Encoder**: Esta parte de la red es la encargada de obtener características a partir del conjunto de datos utilizados. Se puede utilizar los pesos de una red preentrenada para la detección de objetos, como podría ser VGG [26], para obtener estas features.
- **Decoder**: Esta segunda parte es la encargada de convertir las características obtenidas en el encoder en una salida, siendo esta la predicción realizada por la red, formando una segmentación semántica a nivel de píxel.

Respecto al uso de pesos preentrenados, técnica llamada en inglés **transfer learning**, es una técnica que se basa en la utilización de los pesos de una red entrenados sobre un conjunto de datos y utilizar estos como punto de partida para volver a entrenar la red sobre otro conjunto de datos. Por ejemplo, si estamos usando VGG, podríamos utilizar un archivo que contuviera los pesos de la red entrenada sobre un gran conjunto de datos como punto de partida para nuestro próximo entrenamiento.

Esto permitiría tener como punto de partida una red que ya está entrenada para obtener las características más importantes en las imágenes, y poder entrenar nuestra red con un pequeño conjunto de datos, por ejemplo, de 1.000 imágenes. Si se realizase ese entrenamiento desde 0, sin el preentrenamiento, el resultado sería peor que con este. Esto convierte esta técnica en una herramienta muy valiosa a tener en cuenta cuando se quiera realizar el entrenamiento de una red.

3. Estado del arte

Una vez realizada el trabajo teórico necesario para conocer los elementos fundamentales del deep learning, se han elegido las redes que realizan una segmentación semántica como las mejores para conseguir el objetivo de este trabajo. Esto es debido a que, para la detección de elementos como las zonas transitables, nos favorece una selección a nivel de píxel, ya que nos permite exactamente saber la zona transitable, sin elementos extras debidos a la bounding box, como pasaría si hiciéramos la selección basada en el área de estas.

Las redes convolucionales que realizan una predicción basada en la imagen completa se han descartado también, debido a que de las tres opciones es la que menos se ajustan a las necesidades de este proyecto. Ya que estas redes se encuentran muy limitadas para el objetivo de este trabajo.

3.1. Alternativas para la segmentación semántica en el deep learning

3.1.1. Metodología para explorar las alternativas

Para llevar a cabo la elección del mejor modelo a utilizar en este trabajo, se ha realizado una exploración de los siguientes:

- ENet [45]
- SegNet [23]
- FCN [28]
- DeepLab [46]
- PSPNet [33]
- ICNet [24]

Los criterios que se han tenido en cuenta a la hora de elegir el modelo más adecuado han sido el mIoU y el tiempo de procesamiento medio por imagen.

El **mIoU** (mean Intersection over Union) es una métrica que se puede utilizar en las redes que utilizan segmentación semántica para saber su precisión. Esta se obtiene calculando la media de los **IoU** (Intersection over Union) de las diferentes clases a nivel de píxel. La fórmula por la que se rige el IoU es:

$$IoU = 100 * \frac{positivos\ correctos}{(positivos\ correctos) + (falsos\ negativos) + (falsos\ positivos)}$$

Fórmula 18. Cálculo del IoU (Intersection over Union).

Esta representa el acierto medio en la predicción, a nivel de píxel, teniendo en cuenta tres elementos:

- **Positivos correctos**: La predicción en el pixel a comprobar es correcta.

- **Falsos negativos**: El resultado indica que el píxel a comprobar no forma parte de la clase que se está prediciendo, pero realmente este si forma parte de esta.
- **Falsos positivos**: El resultado indica que el píxel a comprobar forma parte de la clase que se está prediciendo, pero realmente este no forma parte de esta.

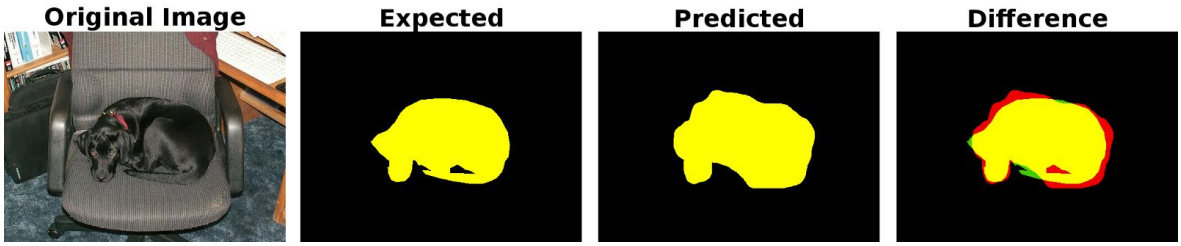


Figura 27. Ilustración de una imagen utilizada para realizar una predicción mediante segmentación semántica, el resultado esperado, el resultado obtenido y la diferencia entre estas, respectivamente. En la imagen correspondiente a "Difference", el amarillo representa los positivos correctos, el rojo los falsos positivos, y el verde los falsos negativos [27].

El IoU tiene un rango de valores entre 0, que representa el peor resultado posible, y 100, que sería una total coincidencia entre el resultado esperado y el resultado obtenido. Es por ello que buscaremos el mayor valor posible de esta métrica.

Finalmente, la razón por la que se han elegido estas dos métricas es que el objetivo de este trabajo es obtener la mejor precisión con el mejor coste computacional. Para así obtener una red capaz de funcionar en robots de bajo coste, los cuales tan solo necesitarían una capacidad moderada de computación, según criterios actuales.

Es por ello que, en la búsqueda del mejor modelo para este proyecto, buscamos la red que tenga la mejor relación entre un mIoU alto y un tiempo de procesamiento bajo.

3.1.2. Arquitectura de las redes a explorar

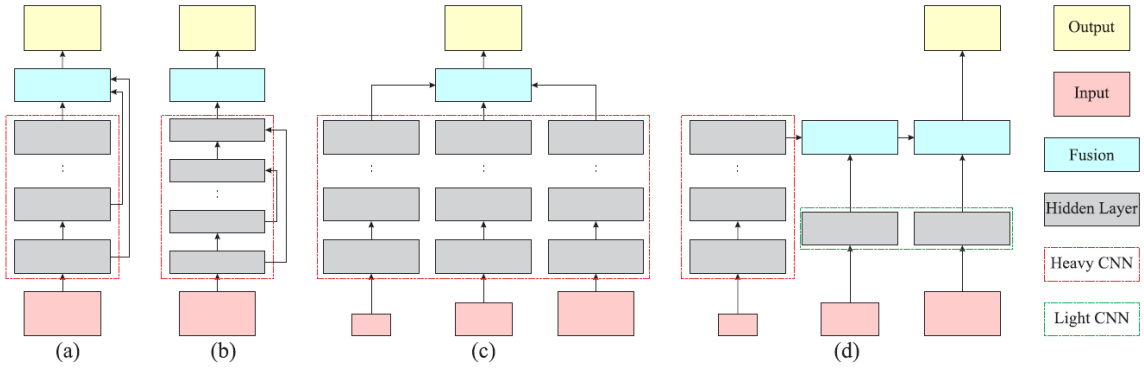


Figura 28. Comparativa de la estructura de las redes FCN (a), SegNet y ENet (b), DeepLab-MSC y PSPNet-MSC (c) y ICNet (d) [24].

En este apartado se explorará la arquitectura de las diferentes redes seleccionadas. Pudiendo así obtener una idea del funcionamiento de las redes de forma más profunda, para así tener una mejor idea del estado del arte actual de las CNNs de segmentación semántica.

Las redes han sido agrupadas como se puede ver en la figura 28, es decir, en los siguientes grupos:

- FCN
- SegNet y ENet
- DeepLab y PSPNet
- ICNet

Esta agrupación se debe a que las redes en un mismo grupo tienen arquitectura parecidas, las cuales se diferencian en el número de capas y parámetros de estas, como por ejemplo, el tamaño de los kernels. Lo cual no afecta para tener un entendimiento de su funcionamiento.

Todas las redes en este subapartado son CNNs, por lo que todas las redes tienen diferentes capas convolucionales que les permite obtener características de la imagen. Estas combinadas con el resto de las capas y los elementos explicados anteriormente son los que permiten a una CNN realizar la predicción final. Debido a que estos elementos son comunes en todas ellas, con diferentes ajustes y cantidad, en este subapartado no se profundizará en estos detalles. En su lugar se explicará únicamente en cada red los elementos que permiten dar como salida una segmentación semántica de la imagen, en vez de una predicción que devuelva una clase, o bounding box, como en otras CNNs.

3.1.2.1. FCN

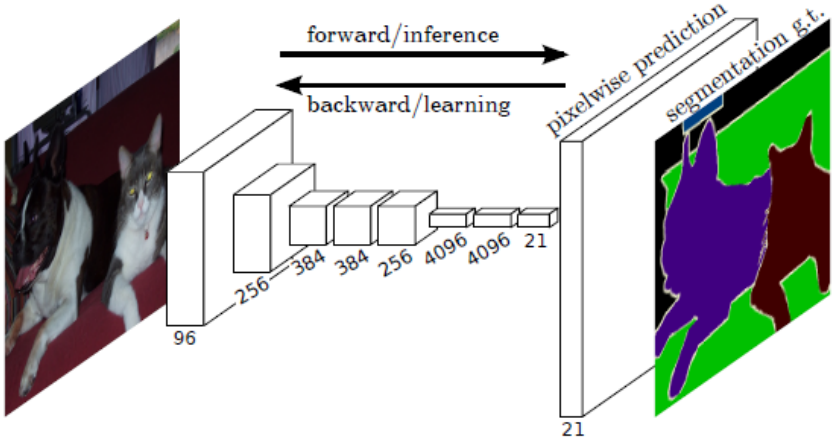


Figura 29. Representación simplificada de la arquitectura de FCN [28].

FCN es una de las redes que tiene un método más simple para obtener la segmentación semántica de la imagen. Esta se obtiene mediante un proceso llamado **Dense Prediction** en el paper de FCN [28].

Este proceso se realiza utilizando upsampling. El upsampling consiste en amplificar la salida de una capa. Esta amplificación se realiza mediante capas de deconvolución en la red.

Las capas de upsampling realizan de diferente forma la deconvolución, dependiendo de que la capa de la deconvolución de como resultado una salida de la red o no, partiendo las capas de deconvolución en capas finales de deconvolución y capas intermedias de deconvolución, respectivamente. En las finales se establece que se realice una interpolación bilinear, mientras que en las intermedias se establece inicialmente la realización de una interpolación bilinear igual a la anterior, pero a partir del entrenamiento de la red el filtro de esta capa se va modificando.

FCN tiene diferentes salidas, siendo estas FCN-32s, FCN-16s y FCN-8s. Los números tras FCN representan la cantidad de upsampling que se ha tenido que aplicar a la entrada de la capa de deconvolución que ha dado como salida esa predicción.

Las diferentes salidas se obtienen a partir de diferentes salidas de otras capas:

- FCN-32s: La salida es obtenida aplicando un upsampling de x32 al resultado de la capa de pooling 5.
- FCN-16s: Para obtener el resultado primero se obtiene la suma de la salida de la capa de pooling 4 y el resultado de aplicar un upsampling x2 a la capa de pooling 5. A esta suma se le aplica finalmente un upsampling x16.
- FCN-32s: Para obtener el resultado primero se suma el resultado de la salida de la capa de pooling 3 y el resultado de aplicar un upsampling x2 a la capa de pooling 5 sumada a la capa de pooling 4. A esta suma se le aplica finalmente un upsampling x8.

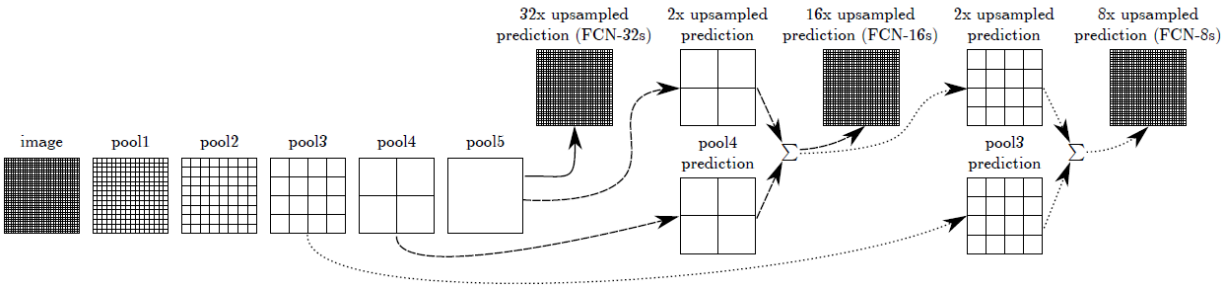


Figura 30. Composición de las diferentes salidas en FCN [28].

Las diferentes salidas darán distintos resultados una vez entrenada la red, pudiendo compararse estas para ver cual obtiene el mejor mIoU.

	pixel acc.	mean acc.	mean IU	f.w. IU
FCN-32s-fixed	83.0	59.7	45.4	72.0
FCN-32s	89.1	73.3	59.4	81.4
FCN-16s	90.0	75.7	62.4	83.0
FCN-8s	**90.3**	**75.9**	**62.7**	**83.2**

Figura 31. mIoU respecto a las diferentes salidas de la red [28].

En la figura 31 se muestra la precisión obtenida por las diferentes salidas de FCN [28]. Pudiéndose comprobar que la salida que más precisión obtiene es la 8s. Por lo tanto, esta será la utilizada para realizar la comparativa de precisión más adelante en este trabajo.

3.1.2.2. SegNet y ENet

SegNet y ENet comparten una estructura similar, estando compuestas ambas redes con una arquitectura de encoder-decoder. La primera parte de la red es el encoder, encargada de extraer características de las imágenes. La segunda es el decoder, encargada de obtener una segmentación a nivel de pixel a partir de las features obtenidas en el encoder.

En este apartado nos centraremos en el caso de SegNet. El encoder de esta red se basa en la arquitectura de VGG-16 con ciertas modificaciones. A partir del resultado obtenido en el segmento anterior trabaja el decoder, que se compone de bloques encargados de realizar un upsampling y después extraer característcas a partir del uso de capas de convolución con activación mediante ReLU y el uso de batch normalization.

SegNet encadena este conjunto de capas creando bloques, como se puede ver en la figura 26. Al principio de cada uno de estos bloques se amplía el tamaño de la entrada hasta llegar al tamaño de la salida. El último de los bloques añade una capa de softmax, que es la que da la salida de la segmentación a nivel de pixel por clases.

A diferencia de FCN, el upsampling realizado por SegNet no es entrenado en ningún momento y no es una interpolación bilineal. SegNet realiza un relleno con ceros en las posiciones en las que no hay un valor establecido al pasar de una entrada a un conjunto de 2x2 valores. Este proceso se encuentra reflejado en la figura 32.

Al ser una matriz de 2x2 y un único valor a establecer, hay que determinar la posición donde se establece este valor. Para saber esta posición se utiliza el índice de la capa de max pooling respectiva en el enconder.

Este sistema se basa en guardar información del encoder, en las capas de pooling, que son de tipo max pooling. Se guarda el índice del valor de mayor tamaño en cada aplicación del filtro. Una vez en el decoder se coge esa posición, y en ella es donde se establece la posición donde se pondrán los valores al hacer upsampling. Esto se puede realizar gracias a la simetría que tiene la red. Ya que en el encoder el tamaño de la salida va disminuyendo inversamente a cómo va creciendo en el decoder.

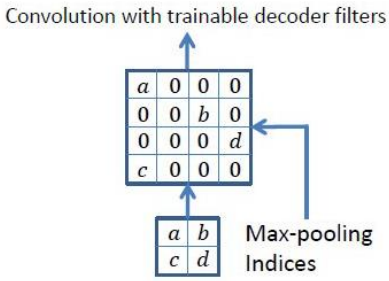


Figura 32. Proceso de upsampling en SegNet [23].

La principal diferencia que se puede observar respecto a FCN, es que esta red empieza a utilizar arquitectura dual: encoder y decoder. En contraposición a FCN, que si bien tenía una estructura de encoder con unas capas finales encargadas de realizar de decodificación, no llegaba a tener una arquitectura dual como SegNet.

3.1.2.3. DeepLab y PSPNet

En este subapartado nos centraremos en PSPNet. Esta red tiene como referencia FCN, poniendo como objetivo superar varios problemas de esta. El principal objetivo a conseguir es el intentar dotar a la red de la capacidad de entender el contexto global de la imagen, pudiendo así poder mejorar la detección gracias a la relación existente entre los elementos de la imagen.

Un cambio importante realizado respecto a FCN es el paso de la estructura de VGG-16 a la de ResNet-101. Este cambio supone una mejora ya que **ResNet** utiliza residual blocks. Estos bloques son un conjunto de convoluciones que son realimentados al final de este por la entrada que llega al inicio del bloque. Esta realimentación dentro del propio bloque permite a las redes llegar a tener una mayor profundidad con una mayor eficacia.

Sobre la arquitectura de ResNet se realizan ciertos cambios para permitir el uso de **dilated convolutions**. Estas son un tipo de convoluciones en las que existe una separación entre los pixeles a los que se les aplica el kernel. Las dilated convolutions tienen un parámetro extra, en este caso indicado como **l**. Este indica el margen superior e inferior para cada posición del kernel al ser aplicado a la entrada.

Por ejemplo, en el caso de una dilated convolution con l=1 no se apreciería una diferencia con una convolución normal. Pero en el caso de l=2, la convolución abarcaría una entrada de 5x5 y aplicaría el kernel sobre las posiciones 1, 3, 5 y de las filas 1, 3 y 5. Este ejemplo se puede observar en la figura 33.

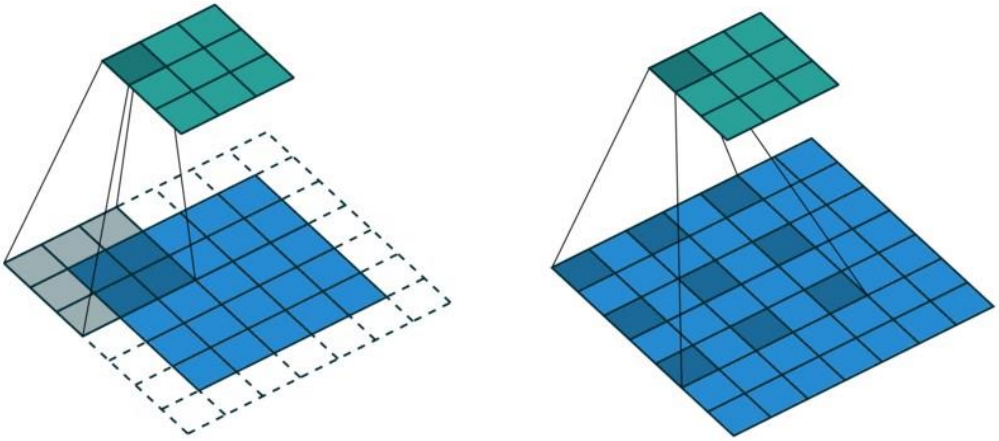


Figura 33. Aplicación de una dilated convolution con l=1 (Equivalente a una convolución normal) a la izquierda. Aplicación de una dilated convolution con l=2 a la derecha.[40]

Otra cambio importante respecto a FCN, introducido para obtener un mejor análisis del contexto de la imagen, es el **Pyramid Pooling Module**. Este módulo se aplica sobre el **feature map** (o **activation map**). Siendo el feature map la salida de una capa de convolución, refiriéndose en este caso a la salida de la última de las capas de convolución antes de llegar al Pyramid Pooling Module.

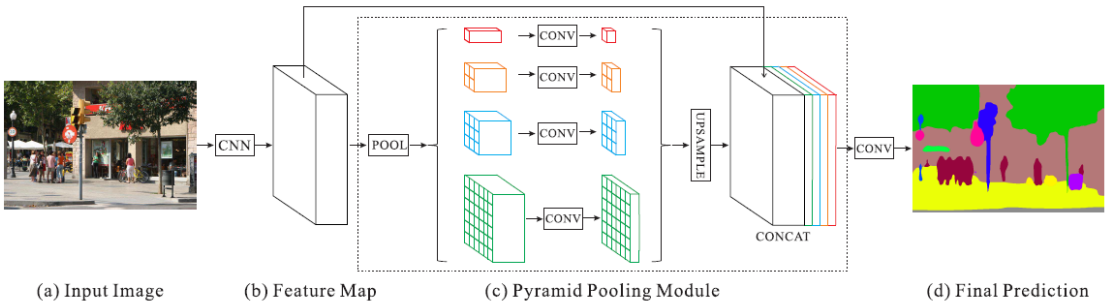


Figura 34. Simplificación de la estructura de PSPNet [33]. Correspondiendo la parte indicada por (b) a ResNet + dilated convolutions.

Esta módulo consiste en la aplicación del average pooling al feature map con 4 tamaños diferentes, dando así 4 salidas:

- 1º Salida: Esta es la salida enfocada a detalles más toscos. Se aplica un average pooling global al feature map, dando un único valor de salida por cada capa del feature map.
- 2º Salida: Se divide cada capa del feature map obteniendo subregiones que parten la imagen en 2x2 subregiones. Después a cada una de estas subregiones se le aplica el average pooling.
- 3º Salida: Se divide cada capa del feature map obteniendo subregiones que parten la imagen en 3x3 subregiones. Después a cada una de estas subregiones se le aplica el average pooling.

- 4º Salida: Este el salida enfocada a detalles más finos. Se divide cada capa del feature map obteniendo subregiones que parten la imagen en 6x6 subregiones. Después a cada una de estas subregiones se le aplica el average pooling.

Tras obtener estas 4 salidas se aplica una convolución 1x1 a cada una de estas salidas con el objetivo de que cada una de esta salidas tenga una dimensión correspondiente a las capas del feature map partido 4. Por ejemplo, si el número de capas del feature map es 2048, cada una de las salidas del Pyramid Pooling Module tendrá 512 capas.

Después se realiza un upsampling para que el tamaño de las diferentes salidas coincida con el tamaño del feature map, basándose este en la realización de una interpolación bilineal, al igual que en FCN. Una vez realizado, se concatenan las capas resultantes de las 4 salidas y el feature map. Finalmente, esta combinación es dada como entrada a una capa de convolución cuya salida es la segmentación semántica obtenida por la red.

También es destacable de PSPNet el uso de un loss auxiliar durante el entrenamiento. Este loss es utilizado para mejorar el proceso de aprendizaje de la red durante el entrenamiento, siendo una salida que se encuentra antes de llegar al final de la red y que durante el proceso de predicción fuera del entrenamiento no es utilizado.

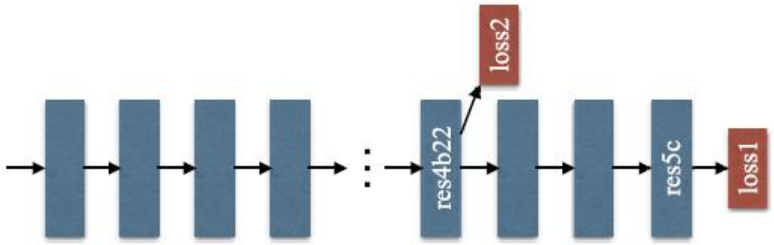


Figura 35. Ilustración del loss auxiliar (loss2 en la imagen) en ResNet101 [33].

Además de los múltiples loss, PSPNet también utiliza múltiples entradas en su variante MSC (multi-scale). Estas entradas son la imagen de entrada original a diferentes escalas. A partir de las diversas entradas se obtienen diferentes features, siendo estas combinadas posteriormente para su uso posterior en la red.

Todos estas mejoras suponen que PSPNet tenga una mejora notable en la precisión de sus análisis respecto a las redes mencionadas en los subapartados anteriores.

3.1.2.4. ICNet

ICNet ha sido diseñada con el objetivo de conseguir un equilibrio entre velocidad y precisión. Ya que los autores consideraban que, si bien se habían hechos grandes avances en los años anteriores en cuanto a la precisión de las redes, se había dejado de lado la búsqueda de un mayor rendimiento.

Para conseguir este objetivo se propone la combinación de la rapidez del análisis de las imágenes de pequeño tamaño con la mejor calidad de la inferencia en imágenes con una

alta resolución. Para ello se crean tres ramas, siendo la primera la más grande de ellas, ya que tiene un mayor número de capas y filtros que las otras. La imagen es dada a la red como entrada con tres resoluciones, la original, reducida a la mitad y reducida a un cuarto, siendo la de menor resolución utilizada como entrada de la rama más amplia, para obtener un resultado que se pueda ir mejorando. Después, mediante el uso de **cascade feature fusion unit** y **cascade label guidance**, se refina el resultado mediante las características obtenidas de las otras dos entradas de mayor resolución.

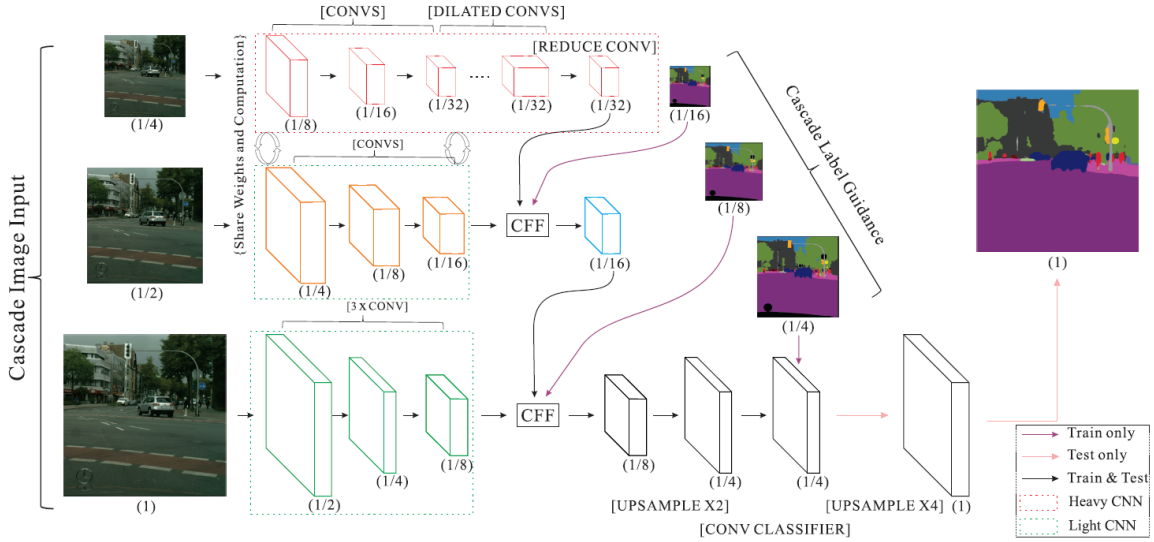


Figura 36. Estructura de ICNet [24].

La implementación de **cascade feature fusion unit** se basa en un bloque de capas y operaciones. Este bloque tiene tres entradas:

- F1: Feature map de tamaño C1 x H1 x W1. Su tamaño es la mitad que el de F2.
- F2: Feature map de tamaño C2 x H2 x W2.
- Label: Ground-truth con dimensiones 1 x H2 x W2.

Primero se le aplica un upsampling al doble de tamaño, a través de interpolación lineal, a F1. Consiguiendo así igualar el tamaño de F2. Después se le aplica a F1 una capa de dilated convolutions con un kernel de tamaño C3 x 3 x 3 con una dilatación de 2 para refinar las features a las que se la ha realizado el upsampling. Obteniendo un resultado con dimensiones C3 x H2 x W2. Después la salida es tratada en una capa de batch normalization, para normalizar el resultado.

A la entrada F2 se le aplica una capa de convoluciones con un kernel de dimensiones C3 x 1 x 1. De esta manera se obtiene que F1 tiene el mismo número de capas (C3) y las mismas dimensiones de estas (H2 x W2). Después la salida es tratada en una capa de batch normalization, para normalizar el resultado, al igual que en F1.

Tras ello, ambas capas son sumadas mediante una suma elemento a elemento y pasada por una función de activación ReLU. Al acabar este proceso se obtiene la salida F2’ con dimensiones C3 x H2 x W2.

La unidad de cascade feature fusion tiene otra salida más, se trata de un loss auxiliar obtenido a partir del upsampling de F1 para mejorar el aprendizaje en este durante el entrenamiento.

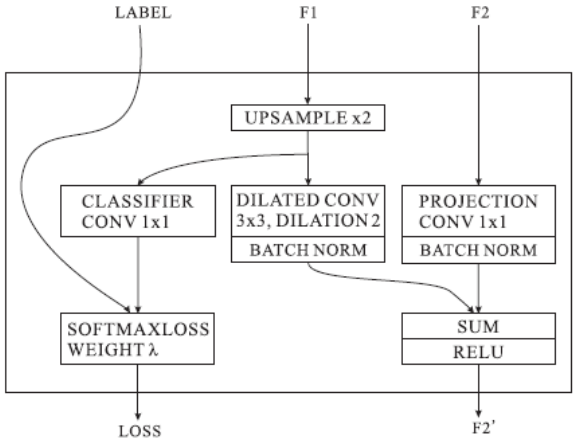


Figura 37. Esquema de cascade feature fusion [24].

La implementación de **cascade label guidance** se basa en la utilización de salidas auxiliares, a la hora de realizar el entrenamiento. Esto quiere decir que existe una salida en cada una de las tres ramas de componen de red: la rama de baja resolución, la de resolución media y la de alta resolución. Estas salidas son el ground-truth a escala 1/16, 1/8 y 1/4 respectivamente, calculándose un loss a partir de cada una de ellas.

Las salidas auxiliares de la rama con ¼ de la resolución y de la rama de con ½ de la resolución son una salida de sus respectivas unidades de CFF (Cascade Feature Fusion). En cambio, la salida de la rama de alta resolución se obtiene tras hacer un upsampling x2 y aplicar una capa convolucional para obtener la predicción de esta rama. Todo este proceso se puede observar en el esquema de la figura 37. Estas salidas auxiliares son desactivadas tras el entrenamiento, utilizándose en su lugar un upsampling x4 de la salida auxiliar de la rama de alta resolución.

Este modelo en cascada, junto a la ligereza de las ramas de media y alta resolución, permiten a ICNet obtener un buen rendimiento mientras mantiene una buena precisión.

3.1.3. Comparativa de resultados

Para poder comparar las redes se han utilizado los valores obtenidos [24] al calcular la precisión sobre el dataset CityScapes [41], así como el tiempo necesario para obtener la predicción por parte de la red. El dato del tiempo necesario para realizar una inferencia por parte de la red es el tiempo obtenido al realizar la operación en una GPU NVIDIA Titan X. A partir de estos datos se ha realizado la tabla 1.

Método	mIoU (%)	Tiempo (ms)	Framerate (fps)
SegNet	57.0	60	16.7
ENet	58.3	13	76.9
DeepLab	63.1	4000	0.25
FCN-8s	65.3	500	2
ICNet	69.5	33	30.3
PSPNet	81.2	1288	0.78

Tabla 1. Comparativa de mIoU, ms y fps de diferentes CNNs que realizan una segmentación semántica.

Los datos se encuentran ordenados según el mIoU obtenido por la red. Siendo la red con mejor resultado PSPNet. Pero solo pudiéndose alcanzar 0.78 fps. Respecto a la red más rápida, esta es ENet, con la que se ha obtenido 76.9 fps, pero en cambio el mIoU es el segundo más bajo, obteniéndose un 24.2% menos que PSPNet.

Al buscarse un equilibrio entre precisión y rendimiento, la mejor opción es ICNet. Ya que es la segunda red con mejor precisión, así como la segunda red más rápida. Además, al superar en las pruebas el margen de 30fps, se abre la posibilidad de su uso para aplicaciones que requieran procesamiento en tiempo real de imágenes. Es por ello, que basándose en los resultados únicamente, esta es la que se ha considerado mejor opción.

3.2. Arquitectura elegida

Un elemento a tener en cuenta a la hora de elegir una arquitectura que se va a usar es si es una arquitectura nueva o ya se encuentra asentada. Este factor es importante, ya que las redes cuya arquitectura lleva más tiempo publicada, tienden a tener un mayor número de implementaciones open source.

Esto nos permite poder comprobar que realmente los resultados obtenidos en el artículo, donde se publicó originalmente la red, se han podido llevar a la práctica. Así como en el caso de querer trabajar con estas, poder empezar con una base funcional y ajustar este código a tus necesidades.

Como se indica en el subapartado anterior, la red que se ha determinado que tiene mejores resultados en relación precisión rapidez es ICNet. Pero esta red es la más reciente de todas las analizadas, lo que se traduce en que en el momento de elegir una red para este proyecto es la que menos implementaciones open source tiene. Además, estas implementaciones no se encuentran tan depuradas como podrían estar las de redes más antiguas.

Esto puede llevar a que su elección pueda originar una mayor cantidad de trabajo práctico, ya que previsiblemente se necesitará invertir más tiempo en su depuración y en ajustes que se quieran realizar para conseguir el funcionamiento deseado. Es por ello que este es un factor a tener en cuenta en la decisión final de la red.

Finalmente, la red elegida ha sido ICNet, ya que como se ha comentado anteriormente, al ser la red más reciente, su uso puede suponer una cantidad de trabajo, pero también es la que muestra unos resultados más prometedores. Por ello, se ha decido asumir este posible trabajo extra, buscando mejorar los posibles resultados finales obtenidos.

4. Herramientas y metodología

En este apartado se tratarán los recursos más importantes que se han utilizado para poder realizar este trabajo. Estos recursos se dividen en los siguientes apartados:

- Lenguajes, librerías y frameworks: Necesarios para la creación de los programas utilizados para realizar este trabajo.
- Herramientas y simuladores: Cubren diferentes necesidades. Estas van desde la gestión y control de versiones del código (BitBucket), hasta el permitirnos tener un entorno donde simular el funcionamiento de un robot móvil a partir del trabajo realizado (Unreal Engine 4 junto a AirSim).
- Entorno de trabajo: Esta sección corresponde al SO y al conjunto de programas utilizados para establecer un entorno de trabajo que permita realizar el desarrollo de este proyecto de forma efectiva.
- Hardware: Se describe el hardware utilizado a lo largo del proyecto. Cubre desde el equipo utilizado durante la creación y comprobación del código necesario para el proyecto hasta el hardware de los equipos utilizados para el entrenamiento y pruebas de rendimiento.
- Metodología: Descripción de como se ha organizado y desarrollado el trabajo a realizar a lo largo del proyecto.

4.1. Lenguajes, librerías y frameworks

4.1.1. Python

Python [29] es un lenguaje interpretado, orientado a objetos y de alto nivel. Su filosofía hace hincapié en la creación de un código limpio que favorezca su lectura y su posterior mantenimiento. También soporta módulos y paquetes, lo que favorece una programación modular y la reutilización de código.

Este lenguaje de programación se encuentra muy extendido en la comunidad científica en general, siendo uno de los lenguajes más utilizados en el campo de la IA. Razón por la cual se ha elegido como lenguaje a emplear, ya que dispone de una gran cantidad de librerías y frameworks a utilizar en este campo.

4.1.2. NumPy

NumPy [30] es una librería fundamental para la computación de problemas matemáticos en Python. Aporta soporte para el uso de arrays multidimensionales, varios objetos derivados (como matrices) y varias funciones para realizar de forma rápida operaciones con estos datos. El núcleo de NumPy es el objeto *ndarry*. Este encapsula arrays n-dimensionales con tipos de datos homogéneos, con varias operaciones siendo realizadas en código compilado por el rendimiento.

Una ventaja que supone la utilización de NumPy es la velocidad que este ofrece al trabajar con matrices y arrays frente a la velocidad que se obtiene utilizando Python puro. Por

ejemplo, en el caso de calcular la desviación media en un millón de datos tipo float en NumPy, frente a Python puro, supone obtener el resultado en 5.97ms frente a los 183ms que se tardaría sin el uso de este [31]. En este ejemplo, el uso de NumPy permite obtener el mismo resultado en aproximadamente el 3.2% del tiempo utilizado en el cálculo en Python puro.

4.1.3. CUDA

CUDA es una plataforma de computación paralela y un modelo de API creado por NVIDIA que se provee como una extensión de **C/C++**. Permite a los desarrolladores usar **GPUs**, que soporten CUDA, para realizar implementaciones paralelas de diferentes problemas.

CUDA requiere el uso de "Computación heterogénea" (Heterogeneous Computing en inglés). Este tipo de computación se refiere a aquella que utiliza más de un tipo de procesador. En el caso de CUDA es utilizada ya que hay código que se ejecutará en la CPU (host code) y código que se ejecutará en la GPU (device code). El programador es quien debe decidir qué parte del código se ejecuta en la CPU o en la GPU.

Una estructura básica de los programas que utilizan esta plataforma sería:

1. Se obtienen los datos, se reserva memoria en la GPU y se lanza el código en esta.
2. La GPU ejecuta el código, el cual suele ser un código que realiza una tarea con un alto coste computacional y que puede ser paralelizada.
3. Se obtiene el resultado obtenido en la GPU a la memoria del host para un posterior procesamiento o almacenamiento.

Aunque esta es la estructura más básica existen otras posibles implementaciones, pero la anterior sirve como ilustración del funcionamiento base.

Un elemento importante de CUDA es la gestión de los hilos donde se realizará la computación. Esta se realiza a través de la jerarquía de hilos (thread hierarchy en inglés), componiéndose de tres niveles, que van de mayor a menor:

1. **Grid**: Grupo de bloques.
2. **Block**: Grupo de hilos.
3. **Hilo**.

El procedimiento que sigue un código para ejecutarse en la GPU, llamado **kernel**, que va desde una función hasta un programa completo, pasa por diferentes fases. Primero este kernel es ejecutado en la GPU, para el cual se crea un grid, donde habrá diferentes bloques que contendrán hilos. En estos hilos es donde se realizan los cálculos necesarios para poder resolver el problema planteado. Esta jerarquía se puede ver en la figura 38.

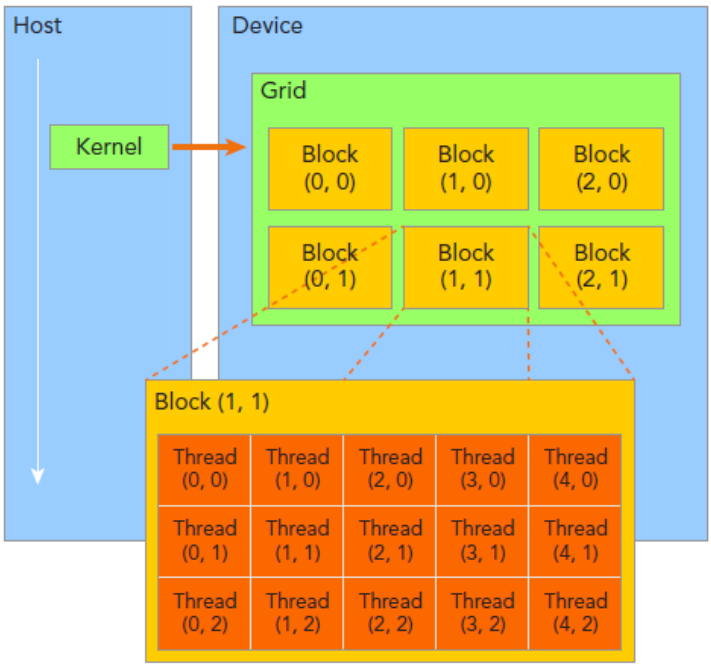


Figura 38. Jerarquía de hilos (thread hierarchy) de CUDA [32].

La ventaja de la utilización de CUDA, y por la tanto las GPUs, es que estas permiten ejecutar muchos más hilos a la vez que un procesador. Esto es debido a que la arquitectura de la GPU se basa en el uso de un mayor número de núcleos (CUDA cores en este caso) pero menos potentes que los núcleos de un procesador. Esto para ciertos problemas paralelizables supone una gran ventaja.

Comparando el número de núcleos de un procesador con los núcleos CUDA de una GPU NVIDIA nos encontramos con una gran diferencia en cuanto al número de estos. Mientras que los procesadores suelen tener de 1 a 16 núcleos en la mayoría de los casos, en el caso de las GPUs podemos encontrarnos que llegan a tener varios miles de cores.

4.1.4. CuDNN

NVIDIA CUDA Deep Neural Network library (CuDNN) [34] es una librería para la aceleración mediante GPU de las operaciones primitivas usadas en las redes que utilizan deep learning, siendo esta utilizada junto a CUDA.

CuDNN da acceso a implementaciones altamente ajustadas para las rutinas estándares como forward propagation, backward propagation, pooling, normalización (normalization en inglés) o capas de activación (activation layers en inglés).

Es por ello que su uso se ha extendido, contando actualmente con soporte en varios framework ampliamente utilizados en el campo del deep learning, como son TensorFlow, Caffe, Caffe2, Keras o PyTorch.

4.1.5. OpenCV

OpenCV (Open Source Computer Vision Library) [35] es una librería open source de software de visión artificial y machine learning que originalmente fue realizada por Intel. OpenCV se creó para proveer de una infraestructura común a las aplicaciones que hacen uso de la visión artificial y para acelerar su utilización en productos comercial.

Esta librería tiene más de 2500 algoritmos optimizados, que incluyen un exhaustivo conjunto de algoritmos tanto clásicos como del estados del arte de la visión artificial y de machine learning. Es una librería muy extendida dentro de su campo y que es usada tanto por compañías como por grupos de investigación. Tiene soporte para ser utilizada en diferentes lenguajes (C++, Python, Java y MATLAB), siendo la versión de Python la utilizada en este proyecto.

Se le ha dado diferentes usos a esta librería en este trabajo, todos ellos relacionados con la manipulación de imágenes y videos. Estos van desde su apertura y creación hasta el tratamiento de estos archivos. Por ejemplo, se ha utilizado para combinar en una sola imagen la salida de la red y la entrada de esta para poder tener una mejor visualización de los resultados.

4.1.6. TensorFlow

TensorFlow es una librería open source para la computación numérica usando grafos de dataflow (dataflow graphs en inglés). Esta es una librería que suele ser utilizada para el desarrollo de proyectos tanto de machine learning como de deep learning. Desarrollada originalmente por Google se encuentra disponible su código de forma pública y tiene detrás una gran comunidad colaborando en su desarrollo.

El uso de dataflow graphs se basa en la creación de un flujo de datos que va transmitiéndose de nodo en nodo, recorriendo el grafo generado. Esto es beneficioso para la creación de redes neuronales y tiene varias ventajas:

- El grafo es portable y puede ser guardado para usar posteriormente en diferentes plataformas como CPU, GPU o TPU. Ayudando además a la computación distribuida a través de las diferentes plataformas [37].
- El grafo es transformable y optimizable. Este puede ser transformado para producir una versión más optima de este para una plataforma en concreto.

El utilizar un frontend cuando se trabaja con TensorFlow, en el caso de este proyecto el frontend de Python, permite abstraerse del uso de dataflow graphs en general, hay casos donde el cambio de paradigma obliga a adaptar la forma de trabajar. Si bien el código se puede escribir en Python utilizando tanto programación imperativa como programación orientada a objetos, existen limitaciones respecto a otros aspectos como, por ejemplo, la depuración.

Estas limitaciones residen en el hecho que trabajando con dataflow graphs no se puede observar el valor de las variables de este en tiempo de ejecución usando alguna utilidad,

como por ejemplo pdb (debugger por defecto de Python). Esto es debido a que primero se define el dataflow graph a través de las diferentes llamadas a TensorFlow y las operaciones se realizan a través de este grafo al iniciar una sesión y ejecutarla. Este funcionamiento impide al debugger ver el valor de las variables a través de una ejecución que se realice línea por línea del código.

Las sesiones de TensorFlow permiten conectar el programa cliente (en Python en nuestro caso) con el C++ runtime, haciendo posible la interacción con el grafo y así trabajar con este. Dentro de esta sesión se encuentra la ejecución actual del grafo y sin esta no se podría interactuar con él.

TensorFlow tiene varias capas, que van desde TensorFlow Distributed Execution Engine que es la que se encuentra a más bajo nivel y es el núcleo de TensorFlow, hasta Canned Estimators que permite trabajar con modelos predefinidos y es el más alto nivel. Entre estas, de menos a más abstracción, se encuentran los Frontends (C++ y Python son los más importantes en nuestro caso) que te permiten trabajar con TensorFlow cómodamente. Los Layers, que permiten crear diferentes capas para nuestro modelo. Y el nivel de Estimators, que permite realizar operaciones como el entrenamiento o la evaluación del modelo abstrayéndose de su implementación.

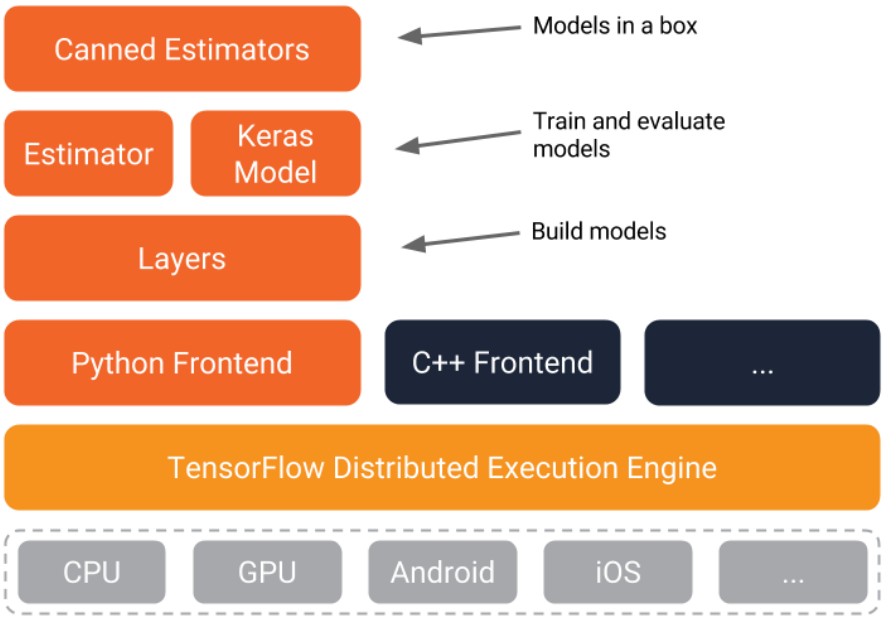


Figura 39. Arquitectura de TensorFlow [36]

Otro elemento importante de TensorFlow son los tensores, ya que son su tipo de dato principal. Los tensores son generalizaciones de los vectores y las matrices para potenciales mayores dimensiones. Internamente, TensorFlow, representa a un tensor como un array n-dimensional con un tipo de dato base, como por ejemplo float de 32 bits o un int de 32 bits.

Los tensores tienen un rango que representa el número de dimensiones de este. Los diferentes rangos representan diferentes entidades matemáticas:

- 0: Escalar
- 1: Vector
- 2: Matriz
- n: n-Tensor

Esta librería es la base de la implementación de ICNet que se usa en este trabajo.

4.2. Herramientas y simuladores

4.2.1. BitBucket

BitBucket es una aplicación web que proporciona un sistema de control de versiones. Este nos permite almacenar nuestro código fuente a través de un historial de versiones, permitiéndonos gestionar, guardar y recuperar los cambios realizados. Permite tanto el uso del protocolo de Git como de Mercurial.

Su utilización ha permitido guardar el código fuente utilizado en un repositorio privado al que únicamente se tiene acceso por nuestra parte mediante el uso de la tecnología de Git. Otra ventaja que nos aporta su uso es la permanencia de un registro de cambios, que ayudan a tener controlados y documentados los cambios realizados en el código a lo largo del proyecto.

4.2.2. Unreal Engine 4 y AirSim

Unreal Engine 4 es un motor gráfico escrito en C++ que se encuentra entre los más potentes actualmente disponibles, siendo capaz de crear escenarios 3D con gran cantidad de detalle. Este motor será el utilizado para crear el entorno de nuestra simulación. Además, se encuentra disponible para Linux con soporte por parte de sus desarrolladores.

Si bien Unreal Engine 4 sirve como base para la simulación, se necesita realizar tareas adicionales como la comunicación con la implementación de ICNet, para poder analizar imágenes procedentes del motor gráfico y mandar una respuesta en base a la predicción obtenida. También se necesita un modelo, que se pueda controlar mediante código, que sea capaz de recorrer el escenario e interactuar con este. Para cumplir estos requisitos se hará uso del plugin AirSim.

AirSim se encuentra desarrollado por Microsoft, estando disponible bajo licencia MIT. Este permite añadir a Unreal Engine 4 dos modelos capaces de ser controlados desde fuera del motor gráfico, existiendo la opción de utilizar Python para ello. El primero es un dron, y el segundo un coche, siendo este último el que se utilizará en la simulación. Además, permite obtener imágenes del motor gráfico a partir de una cámara situada en el frontal del modelo.

4.3. Entorno de trabajo

El proyecto se ha realizado en distintos entornos de trabajo. En cuanto a la parte práctica, la mayor parte del trabajo, que incluye el trabajo de programación, depuración, tratamiento de datos y ajustes de la red, se ha realizado en un ordenador de sobremesa personal. También se ha utilizado un servidor donde se han realizado los diferentes entrenamientos de la red, además de una NVIDIA Jetson TX1 donde se realizó una prueba de rendimiento. Este conjunto de entornos de trabajo diversos ha propiciado el uso de entornos virtuales de Python, donde tener contenidas las dependencias necesarias para el proyecto.

4.3.1. Visual Studio Code

Visual Studio Code es un editor de código fuente que ha sido desarrollado por Microsoft, pudiéndose utilizar tanto en Windows, Linux como macOS. Incluye soporte para varios añadidos, como el control de versiones a través de Git, herramientas de depuración de código en tiempo de ejecución, resaltador de sintaxis o sugerencias de autocompletado. Este es un proyecto open source, que se distribuye bajo licencia MIT.

Este ha sido el IDE elegido para poder realizar el trabajo de programación de este proyecto, debido a que junto al plugin para Python de Microsoft disponible en la tienda de Visual Studio Code, tiene un gran soporte para trabajar con este lenguaje de programación. Este soporte incluye, a parte del resaltador de sintaxis y las sugerencias de autocompletado que suele ser comunes en todos los IDEs de Python, la capacidad de realizar la depuración del código en tiempo real. Siendo esta una característica que no suele estar disponible de forma tan sencilla y cómoda en los IDEs de Python.

4.3.2. Virtualenv

Virtualenv [39] es una herramienta que permite crear entornos virtuales de Python. Estos entornos virtuales son entornos de trabajo aislados que permiten instalar diferentes librerías de código dentro de estas. Por ejemplo, permite tener un entorno de trabajo que requiere un framework, aislándose de otro framework que se utilice y que tenga dependencias en común, pero en otra versión. También sirve para poder crear entornos de trabajo con las nuevas versiones de una librería y comprobar su compatibilidad antes de trasladar el proyecto a esta nueva versión.

En este proyecto ha sido muy útil, ya que al utilizarse diferentes equipos para realizar diferentes tareas y diferentes pruebas de rendimiento permitía tener entornos de trabajo donde solo se tuvieran las dependencias necesarias. Así intentando tener entornos parecidos, para evitar problemas por diferentes versiones de dependencias que ya se encontrasen instaladas anteriormente.

Además, en el caso del servidor utilizado para el entrenamiento tenía una especial relevancia, ya que el servidor era utilizado por diferentes personas, y el uso de un entorno virtual nos aseguraba que no hubiera problemas referentes a las versiones de las librerías necesarias de Python.

4.4. Hardware

Durante el desarrollo de este proyecto se ha utilizado diferentes equipos para cubrir diferentes necesidades, estas van desde el desarrollo del código del proyecto hasta el entrenamiento de la red o el realizar pruebas de rendimiento. Si bien el hardware de los diferentes equipos varía entre ellos, se ha buscado que todos los equipos tuvieran en común la utilización de GPUs de NVIDIA que permitiera utilizar el soporte para CUDA de TensorFlow. Todos estos equipos también tenían en común, si bien no forma parte del hardware, pero afecta en cómo se trabaja con este, el utilizar Ubuntu como SO.

4.4.1. Ordenador de sobremesa personal

Este equipo ha sido en el que se ha realizado la mayor parte del trabajo práctico del proyecto. En él se ha realizado el desarrollo de código, la depuración de este, el tratamiento de datos, los ajustes a la red utilizado, además de varios entrenamientos de la red al inicio del proyecto, debido a la mayor flexibilidad que aportaba para la depuración el trabajar en local. Así pudiéndose observar el flujo de los datos durante el entrenamiento para la depuración y corrección de los errores presentes en este.

Especificaciones:

- CPU: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- GPU: NVIDIA GeForce GTX 1070
- RAM: 16GB DDR3 1600MHz
- SO: Ubuntu 16.04

4.4.2. Servidor

Se ha utilizado un servidor equipado con dos potentes GPUs para el entrenamiento de la red. Aunque para el entrenamiento se ha utilizado únicamente una de ellas, ya que se trataba de un servidor compartido y de esta manera no se utilizaba recursos que no eran necesarios para el entrenamiento. En cambio, se han realizado pruebas de rendimiento sobre ambos equipos.

Especificaciones:

- CPU: Intel(R) Core(TM) i7-6800K CPU @ 3.40GHz
- GPUs:
    - 1x NVIDIA GeForce GTX 1080 Ti
    - 1x NVIDIA GeForce Titan Xp
- RAM: 16GB DDR4 2400MHz
- SO: Ubuntu 16.04

4.4.3. Jetson TX1

Jetson TX1 es el nombre de un ordenador de placa reducida o SBC por sus siglas en inglés (Single Board Computer). Este tipo de hardware permite tener un ordenador completo en una tarjeta que suele ser de tamaño reducido, y generalmente, también tienen un consumo

más reducido que un ordenador de sobremesa. Esto nos permite tener un sistema completo, en un pequeño tamaño y con un consumo menor. Siendo ambas cualidades deseables a la hora de añadir un ordenador al diseño de un robot.

Especificaciones:

- CPU: Quad-Core ARM® Cortex®-A57 MPCore
- GPU: 256-core NVIDIA Maxwell™ GPU
- RAM: 4GB 64-bit LPDDR4 Memory
- SO: Ubuntu 16.04

4.5. Metodología

La metodología utilizada se podría enmarcar dentro de las metodologías ágiles, más concretamente dentro de SCRUM. Ya que tanto el trabajo teórico, como el desarrollo de código, se ha planificado mediante iteraciones, que duraban de una a varias semanas, estableciendo diferentes metas cada una de ellas.

Inicialmente se hizo una planificación tanto del trabajo teórico como práctico que se debía realizar a lo largo del proyecto, y en base a esta se establecieron diferentes metas asociadas a unos plazos para su realización. Tanto las metas como los plazos fueron cambiando conforme avanzaba el proyecto, dependiendo de las diferentes necesidades de este. Estos cambios se debían a diferentes elementos que atrasaban los plazos, como son:

- La gran cantidad de trabajo teórico que se tuvo que realizar durante los primeros meses para tener la base teórica necesaria. Esta cantidad de tiempo se debe principalmente a que el marco teórica de este trabajo pertenece a un campo con el que al comienzo del proyecto se había tenido poco contacto. También resultó relevante que los conceptos aprendidos muchas veces requerían más profundización para su afianzamiento y su mejor entendimiento.
- La dificultad inicial para introducirse en TensorFlow es bastante elevada. Tanto por los conocimientos matemáticos necesarios como por el uso de dataflow graphs, ya que para su utilización se ha necesitado un periodo de adaptación a un paradigma que no era conocido al inicio de este trabajo.
- Profundizar en el conocimiento del tratamiento de imágenes, para ser capaz de realizar todo el proceso que comprende desde el reetiquetado del dataset hasta la depuración de los datos de entrada de la red. Especialmente en lo referente al uso de NumPy y TensorFlow.

5. Segmentación semántica con ICNet

En este apartado se explicará el trabajo práctico realizado para poder llevar a cabo el proyecto. Desde la implementación escogida de ICNet y las modificaciones realizadas a esta, hasta la generación del dataset y la evolución de los resultados obtenidos.

5.1. Código fuente

Para empezar a trabajar con ICNet se ha elegido una implementación open source de la red [42]. Esta implementación se ha elegido ya que se había utilizado **TensorFlow** como base para su implementación, siendo este considerado uno de los mejores framework a utilizar en el campo del deep learning, tras revisar las diferentes alternativas.

Al estudiar el repositorio se encontraron diferentes issues abiertas, realizadas por diversos usuarios argumentando que tenían problemas para hacer funcionar el proyecto de forma adecuada. Por ello, primero se descargó el código y se comprobó que los ejemplos de prueba de los que disponía funcionaban correctamente, así que se siguió considerando a esta la implementación preferente a utilizar.

Estos problemas a la hora de obtener buenos resultados con esta implementación de ICNet también se dieron durante el desarrollo de este proyecto. Estos errores se comentarán más adelante en este apartado.

Para la definición del modelo de la red en TensorFlow se utiliza un sistema de concatenación de capas. A partir de este se puede ir definiendo capa a capa la arquitectura y enlazándolas entre sí. A cada una de estas capas se le puede asignar un nombre, siendo esto útil para la carga de pesos ya existentes en la red, así como para trabajar posteriormente con estas capas.

Utilizando el nombre de la capa se puede señalar cuando se debe cargar unos pesos dados en una capa y cuando no, esto es útil, por ejemplo, al utilizar pesos preentrenados de la red para entrenarla con un número diferente de clases. En esta situación se debería establecer los pesos de las capas de salida que no deben ser cargados, para que así la red pueda editar el número de salidas, inicializando ella misma estos valores.

Esta implementación de ICNet tiene diferentes parámetros a introducir por consola para configurar el entrenamiento. Estos permiten elegir entre las diferentes configuraciones posibles establecidas para la red.

Uno de estos parámetros permite elegir el número de filtros que va a haber en el modelo de la red. Esta elección se realiza mediante el flag **--filter-scale**. El valor 1 de esta opción establece una menor cantidad de filtros, y el valor 2 una mayor cantidad de estos. Esta opción es el modo de realizar en esta implementación el proceso de **model compression** establecido en la especificación original de ICNet. Hay que destacar que en el propio artículo donde se define ICNet se considera que este proceso de **model-pruning** (poda del modelo) reduce notablemente la precisión y hay muchos casos donde su uso no es recomendable.

Aunque en el artículo original de ICNet es establecido como un proceso dinámico de descarte de filtros, en esta implementación es un proceso estático. Ya que en este caso se establece un número menor de filtros al inicio del entrenamiento, en vez de eliminarse ciertos filtros tras acabar el proceso de entrenamiento.

5.2. Generación del dataset

5.2.1. Elección del dataset

La primera elección necesaria era la elección de un dataset base a partir del cual poder trabajar. Este debía estar enfocado a la segmentación semántica, disponiendo de un ground truth que permitiera entrenar y evaluar la red a partir de este.

El dataset más popular para probar redes de segmentación semántica se trata del dataset CityScapes [43]. Este dataset se divide en 30 clases con miles de imágenes y anotaciones. Teniendo una parte de este definida para el entrenamiento y otra para el test. Este primer dataset fue descartado ya que todas imágenes eran captadas en exteriores y por lo tanto no se consideró una buena opción al enforcarse este trabajo en la navegación en interiores.

Del resto de dataset disponibles, el que se encontró más adecuado para los objetivos de este trabajo fue ADE20K [44]. Este dataset disponible de 20.210 imágenes para el entrenamiento de la red (training set) y 2.000 para realizar para validar los resultados de la red (validation set). Estas imágenes se encuentran divididas en 150 clases diferentes, disponiendo cada una de ellas de un ground truth que segmenta la imagen a partir de los diferentes elementos en esta.

Además, ADE20K está compuesto principalmente por imágenes de interiores, aunque existan también imágenes en exteriores, ayudando a que la red pueda realizar una mejor generalización en interiores. Por ello ADE20K fue elegido como el dataset a utilizar para el entrenamiento de la red.

5.2.2. Elección de las clases

Aunque ADE20K está compuesto originalmente por 150 clases, se consideró que una buena estrategia para mejorar los resultados podría consistir en reducir el número de clases. Teniendo como objetivo reducir el número de clases a las mínimas necesarias para poder realizar tareas relacionadas con la navegación en interiores. El criterio inicial fue de utilizar 4 clases: pared, suelo, escalera y obstáculo.

Esta selección de clases fue evolucionando a lo largo del desarrollo del proyecto, acabando finalmente en 3 clases, fusionándose el contenido de la clase pared y escalera. El resultado fueron las siguientes clases:

- Transitable: Compuesto por aquellas clases que representan elementos transitables, como pueden ser el piso de una casa o el pavimento de la calle.
- Intransitable: Clase compuesta por elementos estructurales intransitables, como paredes o escaleras.

- Obstáculos: Clase compuesta por el resto de los elementos clasificados en ADE20K. Todos ellos son objetos y son elementos que normalmente se podrían evitar, bien rodeándolos o con algún otro tipo de acción.

Las clases originales de ADE20K finalmente quedaron organizadas de la siguiente manera:

(1) Transitable	4 (floor), 7(road), 10(grass), 12(sidewalk, pavement), 14 (ground), 29 (carpet), 30(field), 47(sand), 53(path), 55(runway), 95 (soil)
(2) Intransitable	1 (wall), 2 (building, edifice), 6 (ceiling), 9 (windowpane, window), 15 (door), 33 (fence), 44 (sign board), 145 (bulletin board), 43 (pillar)
(3) Obstáculos	El resto de las clases

Tabla 2. Reetiquetado de etiquetas de ADE20K.

Las nuevas clases deben tener asignados un número cada una de ellas. Esto es debido a que a la hora de realizar el entrenamiento, validación y predicción, la salida debe tener indicado en cada valor a que clase pertenece, para así poder realizar correctamente la segmentación semántica. Asignándole a transitable el número 1, a intransitable el número 2 y a obstáculos el número 3.

El cambio de criterio de 4 a 3 clases se debió principalmente a 2 factores:

- Existía una gran descompensación entre la cantidad de casos existentes para el entrenamiento entre la clase “escalera” y el resto de las clases. Esta descompensación podía afectar al proceso de entrenamiento empeorando los datos existentes.
- A nivel práctico no aportaba una información útil extra. Ya que en la mayoría de los casos no había un uso diferente entre el que se le da a la clase “intransitable” y el que se le podría dar a la clase “pared” y “escalera” por separado.

Ambos elementos combinados daban lugar a que se haya acabado valorando como mejor opción la introducción de esta cuarta clase en la clase de “intransitables”. Frente, a la alternativa, de intentar usar técnicas de data augmentation para intentar balancear el dataset.

5.2.3. Tratamiento del dataset

Para poder entrenar ICNet a partir del dataset de ADE20K se utiliza una pareja de imágenes para cada caso, una corresponde a la imagen original, que es una fotografía tomada, y la otra corresponde a una imagen que indica a través de un ground truth los elementos en la imagen original. Si bien existen diferentes anotaciones en ADE20K para cada imagen original, la descrita anteriormente será la utilizada para entrenar la red.

El ground truth es almacenado en imágenes con un solo nivel de profundidad, como las imágenes en escala de grises, donde los elementos de la imagen se encuentran anotados. Esta anotación consiste en introducir un valor en bruto en la imagen que identifique a cada pixel con la clase a la que pertenece el elemento del que forma parte.

Por ejemplo, si en un pixel se encontrase una mesa, esta pertenecería a la clase obstáculo. Por lo que los pixeles que pertenecen a la anotación de la mesa tendrán el valor 1 en la imagen usada como ground truth.

Las imágenes usadas para el ground truth también tiene pixeles con valor 0. Los pixeles con este valor son utilizados para indicar que ese pixel no ha sido anotado y que se deberá ignorar al calcular la precisión de la red.

Para definir la relación entre el archivo que contiene la imagen original y el que contiene el ground truth se utiliza un archivo de texto. Este indica la ubicación de ambas imágenes y su correspondencia.

Para el tratamiento de las imágenes que contiene el ground truth se ha realizado una herramienta. Esta herramienta utiliza OpenCV y NumPy para poder recorrer todas las imágenes del dataset cambiando el valor de los elementos del ground truth.

En cada imagen leída, todos los valores eran modificados en base a las nuevas clases. Por ejemplo, todos los valores 4 (suelo) eran cambiados a 1 (transitable), y así con el resto de los elementos a reetiquetar. Tras cada imagen tratada se añade una nueva línea en un archivo de texto que será el encargado de indicar la ubicación y correspondencia entre imágenes.

Además, se ha desarrollado una herramienta, que también utiliza OpenCV y NumPy, para la visualización del dataset. Para comprobar que se ha tratado y reetiquetado correctamente. Esta herramienta permite mostrar una pareja de imágenes, la original y la que contiene el ground truth, a partir del archivo de texto que indica su correspondencia.

Pero aparte de mostrar la imagen en sí, deja ver los valores de los pixeles de la imagen en el lugar donde se posiciona el ratón. Para así poder comprobar que el valor asignado a los diferentes elementos del ground truth es el correcto.

5.3. Arreglos y mejoras en la implementación

Durante el proceso de entrenamiento de la red se tuvieron que afrontar diferentes problemas. Para depurarlos se realizaron diferentes pruebas y se fueron solucionando los diferentes fallos que se fueron encontrando a lo largo del uso de la implementación escogida de ICNet. En este subapartado se trata el proceso de depuración, los cambios y arreglos realizados al código, así como la evolución de los resultados y la conclusión obtenida a partir de estos.

5.3.1. Mejoras en la visualización de resultados

5.3.1.1. Adición del mIoU al log de TensorBoard

TensorBoard es una suit de herramientas de visualización que viene incluida con TensorFlow. Esta ha sido utilizada para poder visualizar un registro del entrenamiento, que nos permite ver el avance de diferente parámetros a lo largo del entrenamiento.

Dos de los parámetros más importantes, loss (calculado en base al conjunto de datos de entrenamiento) y val_loss (calculado en base al conjunto de datos de validación, sobre los que no se está entrenando) vienen ya añadidos en el log que es generado por el código de la implementación elegida de ICNet.

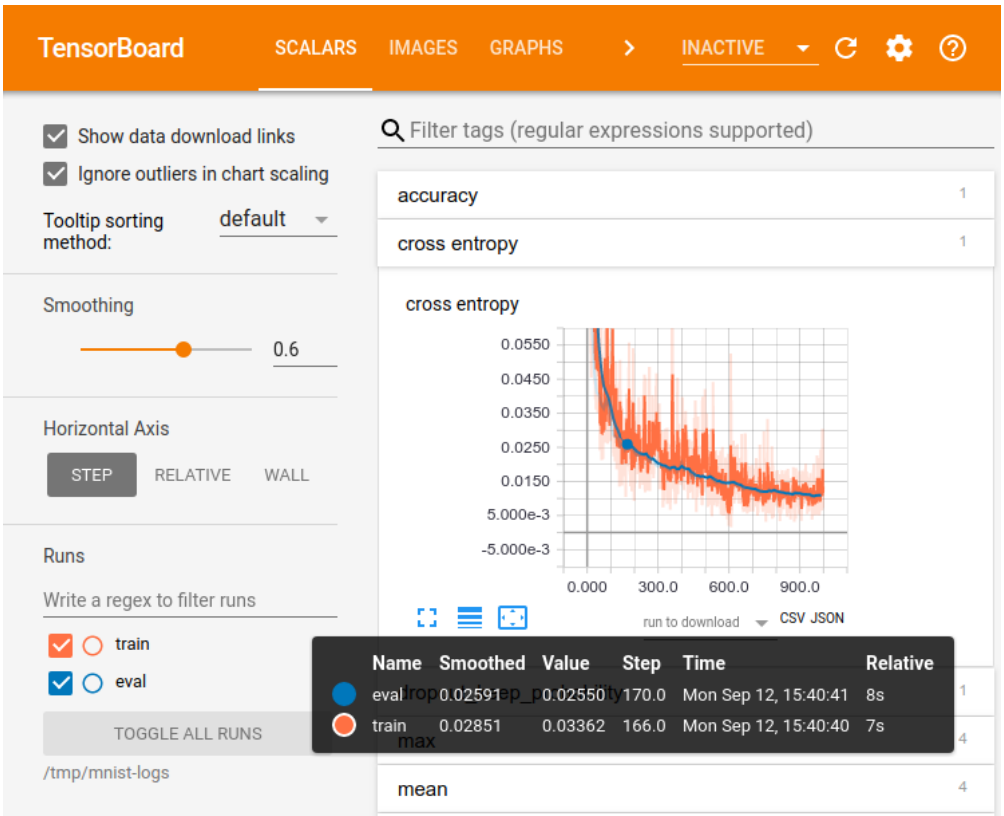


Figura 40. Visualización de un log generado durante el entrenamiento en TensorBoard [47].

Para poder observar el avance de los resultados a lo largo del proceso de entrenamiento, en lo respectivo a la precisión, se añadió el mIoU al log generado durante este. Para poder medir este valor se necesita realizar algunos cambios en la implementación de la red, ya que hace falta editar el proceso de entrenamiento.

Una opción es cargar diferentes instancias de una red en una misma sesión de TensorFlow, a través de opciones como tf.AUTO_REUSE, pero no se optó por esta opción. En su lugar se eligió crear dos instancias de la red, para que el proceso de cálculo del mIoU no estuviera entrelazado con el de entrenamiento.

Para ello se realizaron dos sesiones diferentes, una que se utiliza para el entrenamiento y otra que se inicia al acabar una época (conjunto de minibatches que se necesita recorrer para entrenar sobre todo el conjunto de entrenamiento una vez). En esta segunda sesión se carga el grafo que se acaba de guardar al acabar la época, e inicia una instancia nueva de la red con estos pesos. Se recorren los casos en el conjunto de test, y se devuelve la precisión obtenida, que se guarda en el log.

5.3.1.2. Depuración de la entrada y salida de la red

Debido a que se encontraron diversos fallos que afectaban a la hora de entrenar la red, explicándose estos más adelante, en el proceso de depuración se creó una herramienta para ayudar a esta tarea.

Esta herramienta sirve para seguir el proceso de carga de las imágenes, pudiéndose observar cómo llegan las diferentes entradas a la red para ser utilizadas para el entrenamiento, existiendo a la hora de mostrar la imagen la posibilidad de ver los valores de cada píxel, para así comprobar que los valores que se le dan a la red son correctos.

También se puede mostrar los valores de la imagen del ground-truth una vez tratada y lista para ser establecida como salida esperada. La utilización de esta herramienta resultó útil, ya que permitió encontrar un fallo en el proceso de carga de imágenes que se detalla más adelante.

5.3.2. Problemas encontrados durante el uso de la red

5.3.2.1. Carga de los pesos

La implementación escogida de ICNet permite cargar pesos para la red, especificando la cantidad de pesos utilizados con la opción filter-scale 1 o filter-scale 2, pudiendo así utilizar una red ya entrenada. Pero cuando se trataba de cargar los pesos cambiando la salida para poder volver a entrenar la red a partir de estos existían problemas con ellos.

Con la opción filter-scale 1, se podía cargar los pesos correctamente, descartando los de la última capa de salida para así poder cargar el número de salidas. En cambio, con filter-scale 2, opción que se acabará seleccionando como la mejor para conseguir el objetivo de este trabajo, no se podía realizar este proceso adecuadamente. Esto se debía a que no se descartaban las capas adecuadas, haciendo falta descartar pesos extras.

Esto tiene lugar debido a que durante la carga de unos pesos preentrenados de la red se debe descartar los pesos de la capa de salida, para así poder establecer un nuevo número de salidas. Al cargar los pesos con filter-scale 1, se descartaban la capas necesarias para volver a entrenar la red. Pero con filter-scale 2, solamente se descartaba la salida final de la red. Pero no las dos salidas auxiliares, correspondientes al cascade label guidance de ICNet.

5.3.2.2. Desplazamiento de las etiquetas

Si bien las opciones de configuración establecen que se puede establecer en cualquier número la clase a ignorar, por ejemplo, estableciendo en el valor 255 el valor de los pixeles a ignorar, en la práctica no funcionaba. Esto es debido a que las imágenes se redimensionan al tamaño de entrada, y el relleno necesario para ajustar la imagen a la resolución cuando no coinciden en dimensiones se realiza rellenando con el valor 0. Siendo este relleno producido cuando la imagen es más pequeña que la entrada de la red.

El problema se encontraba en que la red para ajustar estos valores movía en 1 los valores de las etiquetas establecidos pasando de:

- 0: Transitable a 1: Transitable
- 1: Intransitable a 2: Intransitable
- 2: Obstáculos a 3: Obstáculos
- 255: Ignorar a 0: Ignorar

Este problema se debía a un error en la implementación utilizada de ICNet, ya que en la función donde se realizaba la redimensión del ground truth, en los comentarios del código, se establecía que a la entrada se le debía restar el valor de la etiqueta a ignorar, para realizar el relleno con el valor 0 y luego volver a sumar este valor a la entrada. Al realizar este proceso, el relleno y la clase a ignorar acaban con el mismo valor, pero en ningún momento se realiza esta suma en el código de la función. Esta circunstancia da lugar a que el cambio de etiquetas, ya que al final de la función se realiza una conversión a enteros sin signo de 8 bits, se produzca erróneamente al encontrarse con valores negativos, al no haberse producido la suma del valor restado anteriormente.

En este caso en concreto, se daba la situación que al tener las etiquetas [0,1,2,255], ignorándose la etiqueta 255, al producirse la resta se quedaban en [-255,-254,-253,0]. Una vez realizada la conversión al tipo uint8 de NumPy, los valores cambiaban a [1,2,3,0], produciendo el cambio de etiquetas descrito anteriormente.

5.3.2.3. Tratamiento de imágenes ineficiente

En la implementación utilizada se ha optado por utilizar una estrategia de relleno en vez de una redimensionamiento. Esto se puede observar cuando al cargar las imágenes para el entrenamiento, si se da el caso de que las imágenes son menores a la entrada establecida para la red, estas son rellenadas con el valor 0, si se trata del ground truth, o [0,0,0] si se trata de la imagen de entrada, hasta ajustarse a la resolución esperada.

Esta situación no resulta problemática en esta situación, ya que no se pierde información en este proceso. Pero en cambio, cuando la imagen es mayor que la entrada de la red, se sigue sin redimensionar la imagen. En este caso se selecciona un fragmento de la imagen que tenga el tamaño de la entrada de la red, perdiéndose así información para el entrenamiento al recortarse la imagen.

Si bien este no es un error que imposibilite el entrenamiento de la red, sí que puede afectar al resultado obtenido, ya que resulta en una pérdida de información. En su lugar, una mejor estrategia es realizar una interpolación bilineal para ajustar la resolución a la de la entrada o salida de la red manteniendo las proporciones, rellenando con el valor 0 o [0,0,0] en las posiciones que sea necesario.

5.3.3. Resultado inicial de la red

El resultado del primer entrenamiento completo de la red con filter-scale 2 no fueron los esperados. En la segmentación obtenida la mayor parte de la imagen pertenecía a una clase, habiendo pequeñas zonas en su mayoría, perteneciente a otras clases. Una comparativa entre la imagen de entrada y el resultado obtenido se puede visualizar tanto en la figura 41 como en la figura 42.

Figura 41. Primera comparativa entre la imagen de entrada y la salida de ICNet superpuesta a la imagen original (Primer resultado).

Figura 42. Segunda comparativa entre la imagen de entrada y la salida de ICNet superpuesta a la imagen original.

El código de colores elegido para la visualización de resultados en estas imagenes, que corresponden a las salidas cuando el dataset estaba dividido en 4 clases, es el siguiente:

- Verde: Suelo
- Azul: Pared
- Rojo: Obstáculos
- Gris: Escalera

Las clases fueron remodeladas a lo largo del desarrollo del proyecto, por lo que las imágenes de los resultados anteriores el código de colores fue cambiado, ya que las clases fueron cambiadas. Respecto a la diferencia de dimensiones entre la imagen de salida y entrada se debe a la redimensión realizada para ajustar el tamaño de la entrada, por parte de la implementación de ICNet.

La falta de precisión, en los resultados mostrados anteriormente, se debe al conjunto de errores en la implementación que se han comentado anteriormente. Estos se fueron arreglando a lo largo del trabajo con la red. En el siguiente subapartado se muestra los resultados obtenidos al final del trabajo con la red. Pudiéndose comparar los resultados a priori y posteriori de los cambios realizados en el código.

5.4. Resultado final de la red

5.4.1. Proceso de entrenamiento

Durante el desarrollo de este trabajo se realizaron más de una veintena de entrenamientos, muchos de ellos para ver la evolución de los resultados conforme se iban realizando cambios en el código. Pudiéndose observar los resultados de la red tras estos cambios, o el comportamiento de las nuevas funcionalidades implementadas, como la adición del mIoU en TensorBoard.

El entrenamiento mostrado a este subapartado corresponde a un entrenamiento con las siguientes características:

- Se ha entrenado sobre el conjunto de datos de entrenamiento de ADE20K reetiquetado a 3 clases: transitable, intransitable y obstáculos.
- Como datos de prueba, a partir de los que se calcula el mIoU, se ha utilizado el conjunto de imágenes de test de ADE20K reetiquetadas a las mismas 3 clases.
- Batch size = 8. Esto indica que cada iteración corresponde al entrenamiento sobre 8 imágenes y una corrección en base al error calculado a partir de estas 8. Por ejemplo, al realizarse 1000 iteraciones durante el entrenamiento, la red habrá sido entrenada sobre 8000 imágenes.
- Se han realizado un total de 23.300 iteraciones, completándose 9 épocas y habiéndose interrumpido el entrenamiento a mitad de la décima, para así poder comprobar la evolución de los resultados a lo largo del entrenamiento, así como efectos como el del overfitting.

La evolución del loss durante el entrenamiento se puede observar en la figura 43, esta muestra los resultados visualizados desde TensorBoard. Como se puede ver en la imagen, el valor del loss fue decreciendo desde el inicio del entrenamiento, con diversas fluctuaciones, pero manteniendo la tendencia bajista. Esta gráfica nos permite saber que la red está realizando un aprendizaje a partir de los datos de entrada, ajustándose a la información con la que está siendo entrenada.

Sin embargo, no es una métrica adecuada para detectar efectos como el overfitting. Para detectar este, es más adecuada el val_loss (loss sobre el conjunto de datos de validación) o el mIoU. Es por ello, que ambos valores han sido añadidos al log generado y ambas gráficas se muestran en este subapartado.

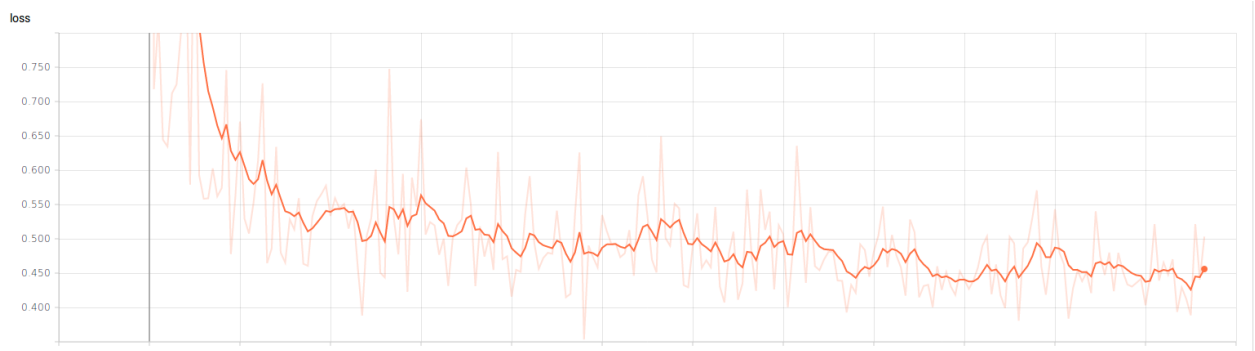


Figura 43. Evolución del loss a lo largo del entrenamiento.

Al observar la gráfica del val_loss, en la figura 44, se puede ver que este valor no sigue el mismo patrón que el loss. Esto es debido al efecto del overfitting, ya que llegado a cierto punto del entrenamiento, la red se empieza a adaptar demasiado al conjunto de datos de entrenamiento, y su capacidad de generalizar empieza a empeorar.

En este caso, a partir de la iteración 8500, indicada con un punto naranja en la gráfica, se puede observar un cambio de tendencia. Hasta ese punto, la tendencia del error es a disminuir, pero tras este empieza un aumento del val_loss. Primero con una sección con un gran aumento, y tras volver a niveles más bajos, aumenta poco a poco la media progresivamente. Estos indicios dan lugar a que se pueda interpretar que a partir de este punto el overfitting empieza a pronunciarse y a ir aumentando.

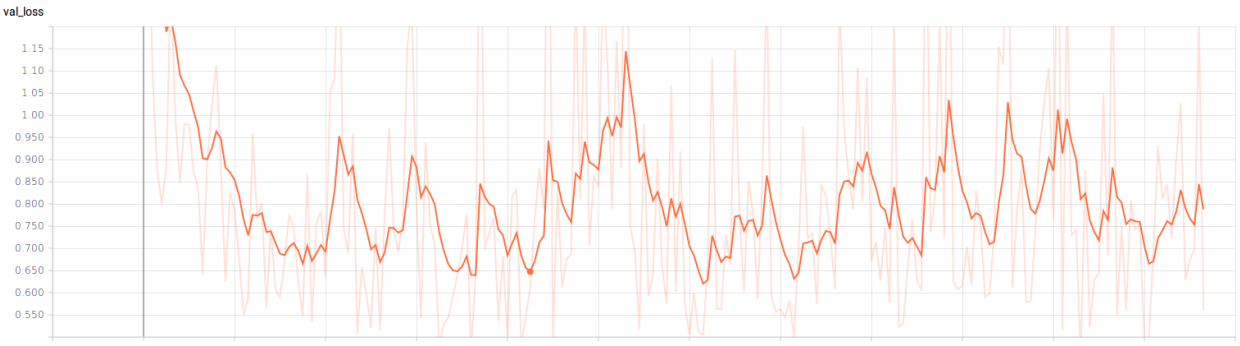


Figura 44. Evolución del val_loss a lo largo del entrenamiento.

El máximo del mIoU se encuentra en las 4.100 iteraciones, con un mIoU de 0.7522. Existiendo otro pico de 0.7501 en 6.800. Disminuyendo a partir de este punto el mIoU poco a poco. Estos datos, junto a los del val_loss, ayudan a reafirmar la teoría que la red a partir de las 8.500 iteraciones ya empieza a sufrir overfitting. Por ello, se ha determinado como mejor opción, el elegir uno de los pesos generados antes de pasar ese número de iteraciones.

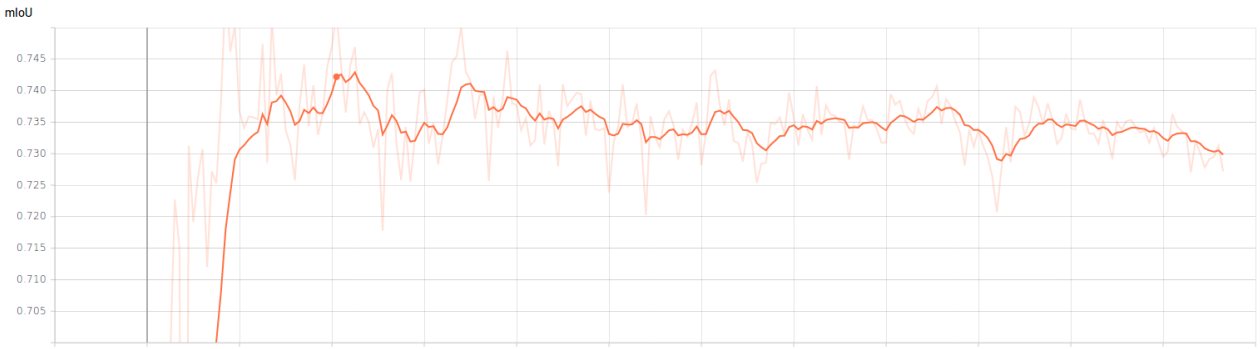


Figura 45. Evolución del mIoU a lo largo del entrenamiento.

Un dato relevante obtenido ha sido que, con el reetiquetado de etiquetas, la red ha obtenido un mIoU de 0.7522 sobre ADE20K. En cambio, sobre el dataset sin reetiquetar, ICNet obtiene 0.3225. Por lo tanto, con el reetiquetado se ha obtenido un mIoU 2.33 veces mayor. Esto es debido a la reducción de la complejidad del problema.

5.4.2. Resultados del entrenamiento

Una vez acabado el entrenamiento, se ha aplicado la red sobre diferentes imágenes para poder ver el resultado obtenido. En estas imágenes se ha utilizado el siguiente código de colores para la visualización de los resultados:

- Verde: Transitable
- Azul: Intransitable
- Rojo: Obstáculos

La figura 46 corresponde a un caso del conjunto de prueba de ADE20K. La inferencia se ha realizado a una resolución de 480x480, y se pueden obtener diferentes conclusiones a partir de esta:

- En el resultado se puede comprobar que existe un conocimiento general, por parte de la red, de la composición de la imagen y de las clases establecidas, aunque este no sea perfecto.
- Existen diversos fallos en la predicción:
    - Pequeñas regiones con una predicción errónea.
    - Predicción inexacta de los extremos de los objetos más finos.
    - Definición inexacta del fin del suelo, perteneciente a la clase transitable.

Figura 46. Primera comparativa entre la imagen de entrada y la salida de ICNet superpuesta a la imagen original (Primer resultado).

Haciendo balance del resultado y las conclusiones obtenidas a partir de este, se considera que es un resultado satisfactorio, pero debido a que esta imagen pertenece al conjunto de prueba puede existir un sesgo. Debido a que al haberse escogido los mejores pesos a partir del mIoU se puede dar la circunstancia de que la red sea buena sobre los datos de entrenamiento, y los de prueba, al haber escogido el mejor de los casos para los datos de prueba. Esto puede significar que para imágenes no perteneciente a este conjunto de prueba los resultados sean peores. Es por ello, que se ha buscado utilizar también imágenes no pertenecientes a este.

En el caso de la figura 47, la imagen representada en esta no pertenece a ADE20K, pudiendo esta prueba ayudar a comprobar si la red tras el entrenamiento no se encuentra demasiado ajustada a los casos que existen en ADE20K.

Figura 47. Segunda comparativa entre la imagen de entrada y la salida de ICNet superpuesta a la imagen original.

El resultado obtenido es bastante similar al del caso anterior, con unos problemas de detección bastantes similares, y un resultado bastante satisfactorio.

Por último, se ha realizado dos pruebas más sobre imágenes no pertenecientes al dataset ADE20K. Además, estas imágenes se han elegido ya que representan un entorno industrial, con elementos diferentes a los existentes en ADE20K, ya que este dataset principalmente contiene imágenes de interiores de casas o de pasillos. En cambio, en las figuras 48 y 49 se pueden observar objetos como palés, conos o una transpaleta.

La primera de estas figuras está compuesta por tres imágenes a su vez. Siendo respectivamente, la imagen de entrada, la segmentación obtenida y una superposición de ambas.

El resultado obtenido es realmente satisfactorio. Ya que se trata de un entorno nunca visto, con objetos tampoco vistos anteriormente, pero se sigue teniendo un entendimiento del entorno con un buen resultado final. De este primer caso se puede obtener las siguientes conclusiones:

- El resultado es satisfactorio, ya que en general, la separación de los elementos es correcta y bastante precisa en cuanto a su contorno.
- El principal fallo que existe es a la hora de clasificar las ventanas. Sobre todo, la que se encuentran al fondo de la imagen. Ya que detecta parte de estas como pertenecientes a la clase “obstáculo”.

En cuanto a los objetos existentes en la imagen, se puede comprobar que son detectados de forma correcta generalmente. Teniendo el principal problema con aquellos elementos más finos. Por ejemplo, se puede comprobar que elementos nunca vistos como la transpaleta o el cono son detectados correctamente.

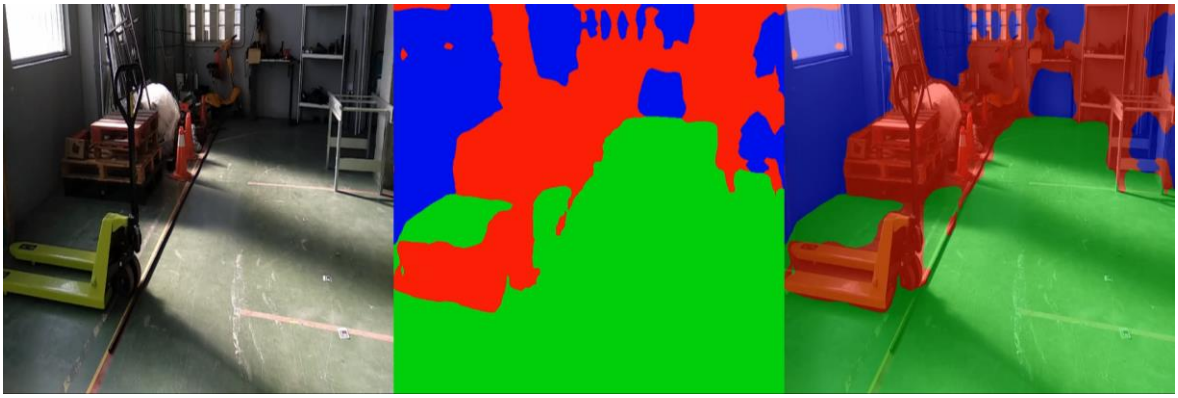


Figura 48. Resultado en un entorno industrial con elementos no vistos anteriormente. Imagen de entrada, segmentación resultante y superposición de ambas imágenes, respectivamente.

La siguiente imagen de este entorno se puede utilizar para observar que hay casos en los que no se obtiene una predicción tan buena. Pero a pesar de ello, sigue habiendo un

entendimiento general del contexto y una clasificación de los elementos en las diferentes clases de forma satisfactoria, pero con mayor cantidad de fallos.

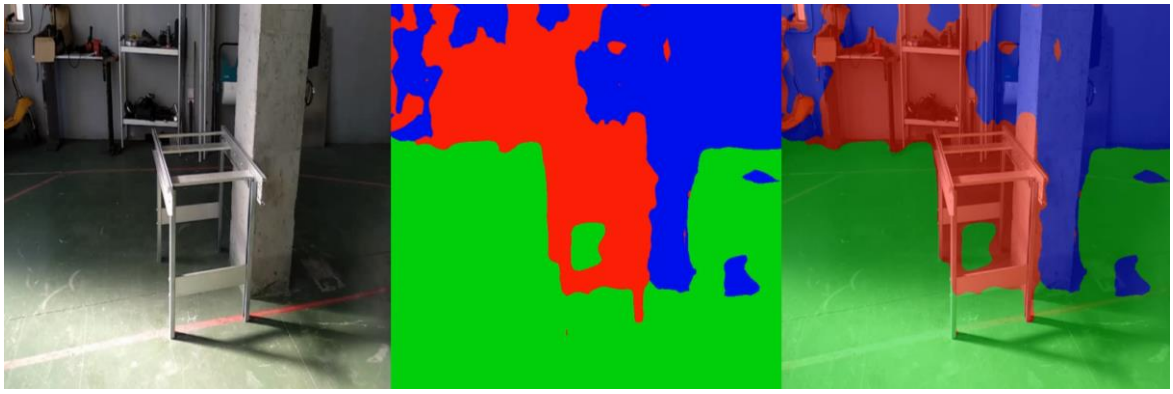


Figura 49. Resultado en un entorno industrial con elementos no vistos anteriormente. Imagen de entrada, segmentación resultante y superposición de ambas imágenes, respectivamente.

Como se puede observar, los fallos de predicción presentes en este último caso son una profundización de los problemas anteriores. Regiones aisladas con predicciones no correctas, pero de dimensiones mayores. No definición exacta del contorno de los elementos, o extremos finos de elementos que no han sido clasificados correctamente.

5.5. Pruebas de rendimiento

Para poder comprobar el rendimiento de la red se han realizado dos tandas de pruebas. En la primera tanda de pruebas, se han utilizado diferentes entradas para la red con resoluciones cuadradas. Para poder utilizar después los resultados obtenidos para comprobar sobre que hardware se puede obtener los fps necesarios para nuestro objetivo. Así como comprobar cómo evoluciona el tiempo necesario para la inferencia en base a la resolución de la entrada.

Equipo	Resolución	Tiempo (s)	Framerate (fps)
Servidor (GTX 1080 Ti)	640x640	0.028	35.71
Servidor (GTX 1080 Ti)	480x480	0.018	55.55
Servidor (GTX 1080 Ti)	320x320	0.017	58.82
Servidor (GTX 1080 Ti)	224x224	0.013	76.92
Servidor (Titan XP)	640x640	0.028	35.71
Servidor (Titan XP)	480x480	0.017	58.82
Servidor	320x320	0.017	58.82

(Titan XP)			
Servidor (Titan XP)	224x224	0.013	76.92
Ordenador personal (GTX 1070)	640x640	0.033	30.30
Ordenador personal (GTX 1070)	480x480	0.021	47.61
Ordenador personal (GTX 1070)	320x320	0.015	66.66
Ordenador personal (GTX 1070)	224x224	0.013	76.92
NVIDIA Jetson TX1	640x640	2.805	0.35
NVIDIA Jetson TX1	480x480	1.685	0.59
NVIDIA Jetson TX1	320x320	0.886	1.12
NVIDIA Jetson TX1	224x224	0.526	1.9

Tabla 3. Primera comparativa de rendimiento.

En base a los resultados de esta primera tanda de pruebas se puede obtener las siguiente conclusiones:

- La velocidad necesaria para cumplir con el objetivo de este trabajo se puede obtener utilizando tanto una GPU NVIDIA GTX 1070, NVIDIA GTX 1080 Ti o NVIDIA Titan Xp. Ya que se considera que al superar la barrera de los 30 FPS se cumplir el objetivo de este trabajo con esa GPU.
- No se consigue el objetivo con el SBC NVIDIA Jetson TX1. Ya que el mejor resultado obtenido se aproxima a los 2 FPS. Menos de 15 veces nuestro objetivo.
- Una reducción a la mitad del alto y ancho (640x640 frente a 320x320), es decir, reducir a ¼ la resolución, de media supone necesita un 60% del tiempo de inferencia original. Por lo que se deduce que la resolución tiene un gran impacto en el tiempo de inferencia necesario.

Después, en aquellos equipos donde se ha superado los 30 FPS en alguno de los apartados, se ha realizado una prueba con diferentes resoluciones que se podrían utilizar como entrada. Siendo 1280x704 el ajuste de la resolución HD (1280x720), ya que la entrada debe ser un múltiplo de 32. Y siendo 640x354 el ajuste de la resolución 640x360 (un cuarto de la resolución HD).

Estas resoluciones fueron elegidas, en el caso de 1280x720 y 640x480, por ser resoluciones estándar, y 640x360 por mantener el ratio de aspecto de la resolución HD, pudiéndose hacer un reducción de la imagen de entrada hasta esta.

Equipo	**Resolución**	**Tiempo (s)**	**Framerate (fps)**
Servidor (GTX 1080 Ti)	1280x704	0.042	23.80
Servidor	640x480	0.024	41.66

(GTX 1080 Ti)			
Servidor (GTX 1080 Ti)	640x352	0.018	55.55
Servidor (Titan XP)	1280x704	0.042	23.80
Servidor (Titan XP)	640x480	0.022	45.45
Servidor (Titan XP)	640x352	0.018	55.55
Ordenador personal (GTX 1070)	1280x704	0.055	18.18
Ordenador personal (GTX 1070)	640x480	0.026	38.46
Ordenador personal (GTX 1070)	640x352	0.022	45.45

Tabla 4. Segunda comparativa de rendimiento.

En base a la segunda tanda de pruebas de puede comprobar que en las tres gráficas se consigue el objetivo de los 30 fps en las tres gráficas para las resoluciones de 640x480 y 640x352. Consiguiéndose obtener 23.80 a resolución HD en las GPUs NVIDIA 1080 Ti y NVIDIA Titan Xp. Abriéndose la posibilidad de estudiar el uso de resoluciones más cercanas a HD si fuera necesario.

5.6. Análisis de los resultados

En lo referente a velocidad como precisión, se ha obtenido a un resultado satisfactorio, ya que con una gráfica NVIDIA GTX 1070, que tiene un coste inferior a los 400€ a fecha de este trabajo, se ha conseguido poder trabajar con las resoluciones de 640x480 y 640x360 a más de 30 fps. Esto abre la puerta a su utilización en proyecto que requieran un procesamiento en tiempo real.

Los resultados también muestran que un posible camino para obtener una mejora de precisión es la reducción de clases, permitiendo un aumento de la precisión. Si bien se pierde la diferenciación entre diferentes tipos de objetos, hay diversos usos que no requieren esa diferenciación, como el caso de la detección de obstáculos en el camino del robot.

Aunque los resultados son positivos, hay que tener en cuenta que no se han contemplado factores como el consumo eléctrico o el tamaño. Siendo estos factores importantes para diversas implementaciones en robots. Ya que estos factores pueden afectar a la autonomía, si se tiene un elevado consumo eléctrico, o pueden dar problemas a la hora del diseño por limitaciones de espacio.

Estos factores no se han estudiado por las limitaciones de tiempo y presupuesto de este proyecto. Pero serían elementos a considerar para una posible ampliación de este. De esta

manera habría que explorar soluciones alternativas, como el uso de una, o varias, unidades de Intel Movidius, Jetson Nano o Google Dev Board, además de estudiar el resto del hardware necesario, como la CPU o la memoria. Estas posibilidades son tratadas con mayor profundidad en el apartado 7, en el subapartado de líneas futuras de investigación.

6. Simulación en Unreal Engine 4

Para comprobar algunos de los posibles usos para la navegación en interiores usando segmentación semántica se ha realizado una simulación en Unreal Engine 4 junto al plugin AirSim. Primero se ha recreado un escenario que sirviese para poder realizar diferentes pruebas, y después se ha diseñado un sistema para controlar el modelo proporcionado por AirSim en base a las predicciones realizadas por ICNet. Siendo las predicciones obtenidas a partir de las imágenes de una cámara situada en el frontal del modelo que se encuentra en el motor gráfico.

6.1. Diseño del escenario

Unreal Engine 4 dispone de una interfaz completa que nos permite realizar todas las funciones necesarias para diseñar nuestro escenario. Desde esta se puede colocar e inspeccionar los elementos que se encuentran en la escena.

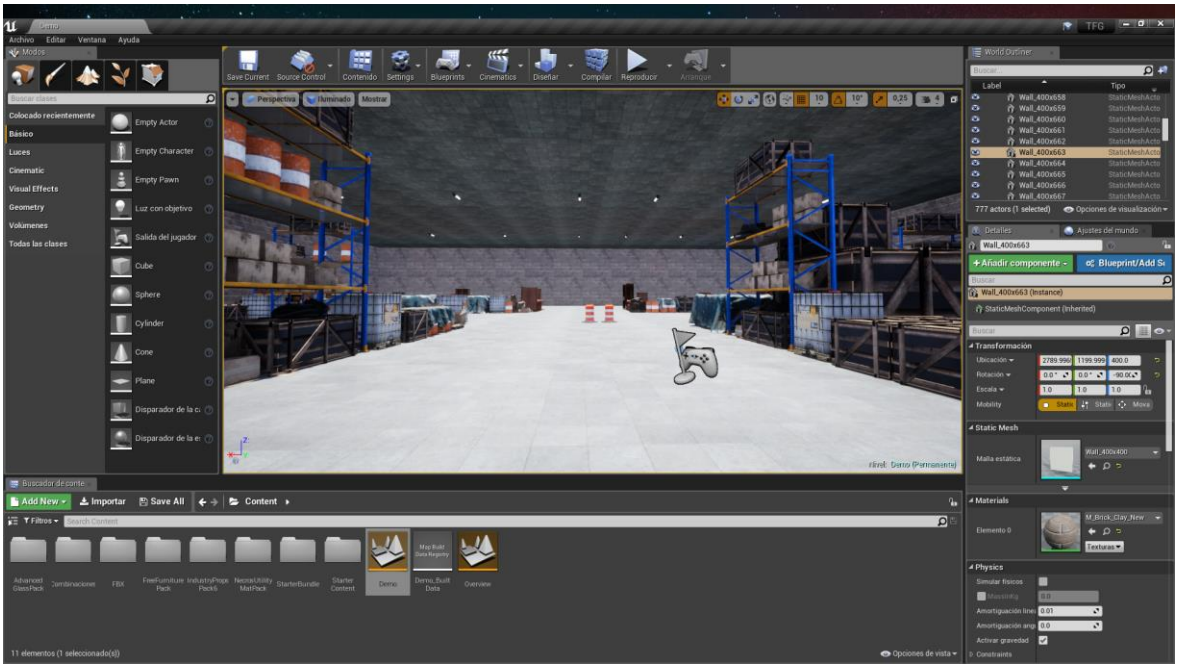


Figura 50. Interfaz de Unreal Engine 4.

El diseño del escenario se basa en dos áreas, con diferentes caminos entre ellas. La motivación de la creación de ambas zonas es que se ha querido colocar elementos diferentes y comprobar la reacción de la red ante cada uno de estos. La primera está compuesta con elementos de carácter industrial, como aquellos que se podrían encontrar en una nave industrial, mientras que la segunda está compuesta por elementos de carácter doméstico.

Los caminos que se encuentran entre las zonas están pensados como elementos de delimitación, así como un medio para que el modelo del coche, proporcionado por AirSim para la simulación, pueda recorrerlos y avanzar por ellos.

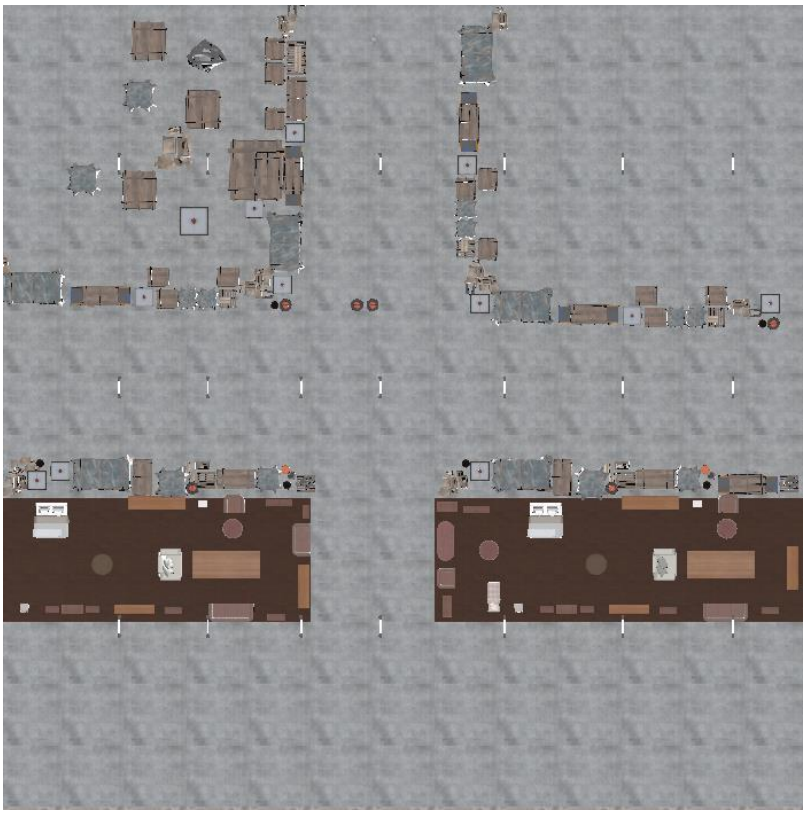


Figura 51. Perspectiva aérea del escenario.

La zona industrial se encuentra compuesta por diferentes elementos, como son pales, estanterías con diferentes objetos, paquetes de madera de grandes dimensiones, cajas de cartón, diferentes tipos de barriles, así como elementos tapados por una lona de plástico.

Figura 52. Vista de zona industrial

Figura 53. Vista de zona industrial

En las dos subzonas que está dividida esta área se puede observar que en una se han añadido elementos tras la primera fila de objetos, mientras que en otra no. Esto se ha realizado para poder ver la incidencia que tiene que los objetos se encuentren directamente delante de la pared, o en su lugar se encuentren seguidos de otros objetos, en la predicción realizada por la red.

Los modelos que se pueden observar en la escena se han obtenido mediante el market de Unreal Engine 4, que da acceso a assets gratuitos, y también provienen de los proporcionados por el propio motor gráfico. Estos se han combinado de diferentes modos para crear conjuntos distintos, permitiendo aportar más variedad dentro de la limitada cantidad de modelos de los que se disponía para hacer la simulación.

Figura 54. Diferentes combinaciones de los modelos.

La segunda zona empieza con un conjunto de elementos de carácter industrial, para poder comprobar la predicción realizada sobre estos cuando los elementos se encuentran de frente en vez de en los laterales, y continua con una distribución en forma exposición con los elementos domésticos. Estos modelos pertenecen a diferentes tipos de estanterías, sofás, sillones, camas y mesas, siendo el suelo también cambiado por uno que recrea la madera, para poder diferenciar la zona y poder observar el comportamiento de la red ante este cambio.

Figura 55. Zona de transición entre elementos industriales y domésticos.

Figura 56. Zona de transición entre elementos industriales y domésticos.

La disposición respecto al área de elementos industriales es diferente, ya que se ha buscado colocar estos con una disposición más parecida a la que podrían tener en una casa. Dejando cierto espacio entre ellos, y orientándolos en relación a los otros. Por ejemplo, colocando los sillones mirando hacia las mesas.

Figura 57. Zona con elementos domésticos.

El punto de salida del escenario se encuentra al inicio de la zona industrial, habiéndose colocado barriles en la vía que permite atravesar ambas áreas, utilizándose estos durante la simulación como obstáculos.

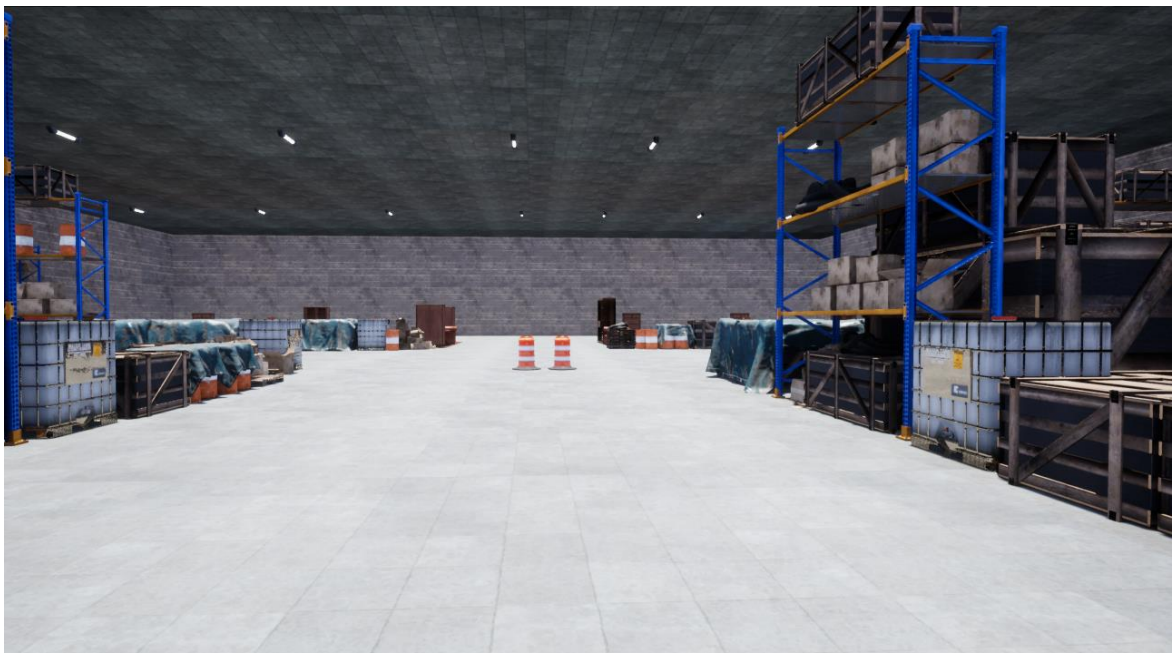


Figura 58. Punto de salida de la simulación

6.2. Simulación

Para poder realizar la simulación se ha planteado un sistema de control basado en la interpretación de diferentes áreas de la segmentación realizada por ICNet, controlando los movimientos a realizar en base a la clase existente en esa porción de la predicción.

El objetivo a conseguir por el modelo que simula nuestro robot es el de recorrer el escenario de principio a fin, esquivando los diferentes elementos que se encuentre por el camino, siendo estos cambiados entre diferentes ejecuciones para comprobar la capacidad del sistema de adaptarse a cambios.

6.2.1. Implementación del sistema de control

Para poder comprobar en tiempo real el funcionamiento del sistema de control se ha realizado una interfaz básica basada en un conjunto de imágenes que muestran los diferentes elementos que se están comprobando a cada frame analizado.

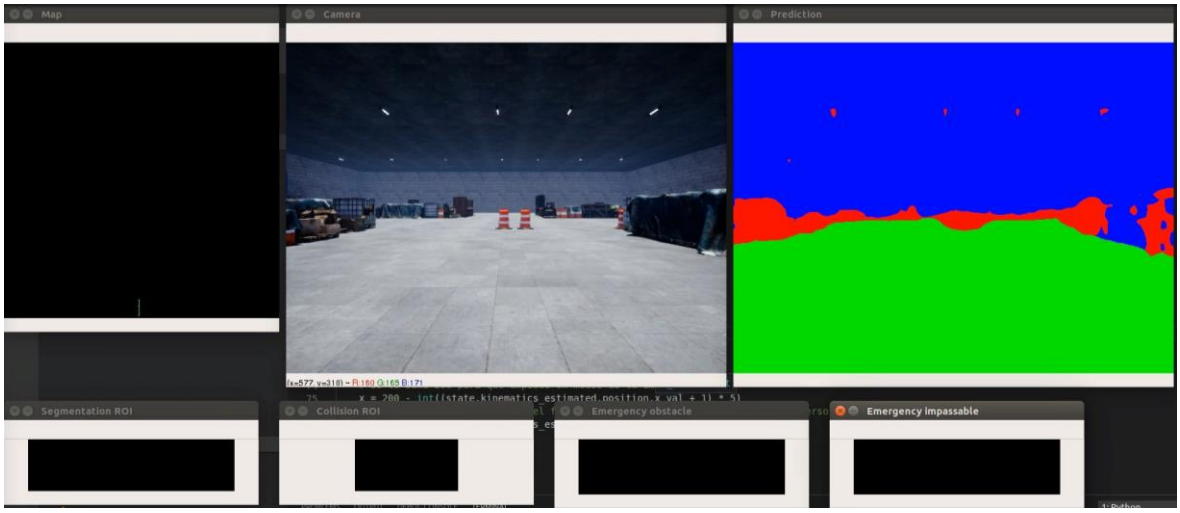


Figura 59. Interfaz de la simulación compuesta de diversas ventanas

Esta interfaz está compuesta por 7 ventanas, cada una de ellas destinadas a una funcionalidad diferente. En los siguientes subapartados se explicará cada una de las vistas que se pueden ver en la figura 59, así como el funcionamiento interno dentro del sistema de control de la parte que muestran. Por ejemplo, al explicar la vista con el título “Collision ROI” se explicará cómo los datos de que son mostrado en esta vista son usados internamente para calcular cuando hay un elemento con el que se podría colisionar si no se corrige el rumbo. Finalmente, en el subapartado 6.2.2 se explicará el sistema como un conjunto, exponiendo diferentes ejemplos de su ejecución, para una mejor comprensión de este.

6.2.1.1. Vista de la cámara y la predicción

En la parte superior de la interfaz se pueden visualizar dos vistas que pertenecen a la imagen capturada desde el motor del juego, desde la cámara situada en el frontal del modelo utilizado, y al resultado de la segmentación realizada por parte de ICNet.

La primera es utilizada para poder conocer en tiempo real frente a qué elementos se encuentra nuestro modelo, así como para poder comparar la entrada a la red con la predicción obtenida por parte de esta. Las imágenes se obtienen gracias a AirSim, que permite realizar peticiones a Unreal Engine 4 desde Python para que este envíe a través de un socket las capturas.

La segunda muestra la segmentación obtenida, que en esta vista se muestra sin ninguna modificación respecto a la dada por ICNet, pero que en otras vistas será tratada en diferentes áreas para realizar decisiones en base a esta.

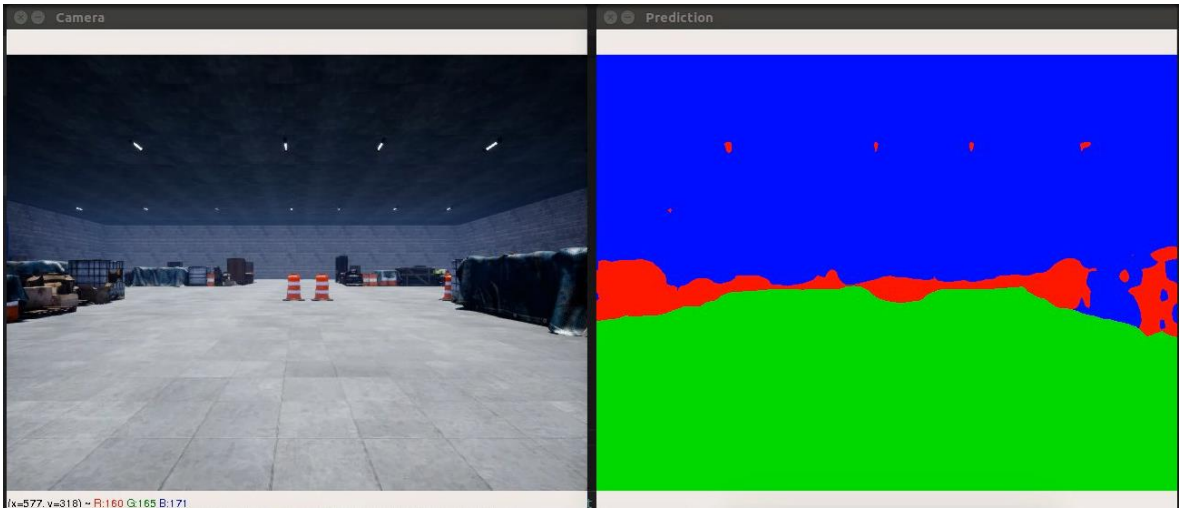


Figura 60. Visualización de la salida obtenida del motor gráfico y la segmentación realizada por ICNet.

6.2.1.2. Mapa del recorrido

La vista en la zona superior izquierda corresponde a un mapa donde se representa el recorrido a lo largo de la simulación. Este mapa es generado a partir de los estados en los que se encuentra en cada momento el sistema de control, así como la posición en la que se ubica el modelo, avanzando a cada iteración del algoritmo. Para poder diferenciar visualmente los cambios en los estados, a lo largo del total de la simulación, se utiliza un color para cada uno de ellos.

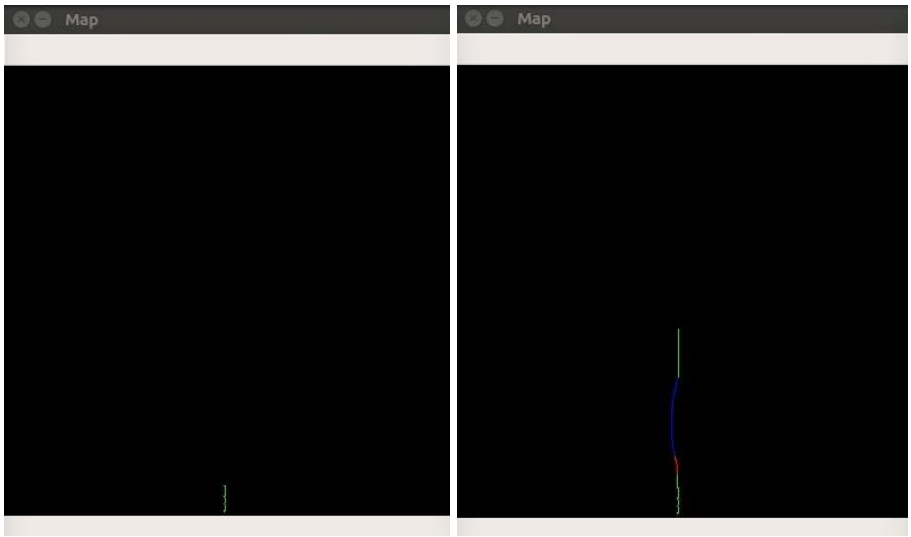


Figura 61. Comparativa entre el mapa generado al principio de la simulación (izquierda) y una vez esta llegando a su fin (derecha).

El color verde en la mapa representa cuando se está siguiendo una trayectoria recta, siendo este el estado inicial. El rojo es utilizado cuando la trayectoria está siendo cambiada para evitar un obstáculo, mientras que el azul representa cuando se está volviendo a la trayectoria original.

6.2.1.3. Vistas de la detección de obstáculos

Las vistas de "Segmentation ROI" y "Collision ROI" están basadas en dos áreas de la segmentación obtenida de ICNet, siendo elegidas ya que las posiciones del eje Y que abarcan son adecuadas para detectar cuando un elemento se está acercando demasiado. Si bien ambas áreas están formadas por los mismos valores del eje Y, en el eje X abarcan diferentes valores. Esto es debido a que "Collision ROI" está pensado para detectar elementos con los que podría chocar el modelo, mientras que "Segmentation ROI" tiene el objetivo de detectar cuando un objeto esta fuera de un rango que pueda implicar una colisión. Por ello, "Collision ROI" abarca un menor rango de valores en X que la otra vista. La palabra **ROI** en las vistas hace referencia a que es una región de interés (**Region Of Interest** en inglés), es decir, que hay elementos relevantes en esa zona.

Para hacer más fácil y eficiente el proceso de detección de elementos en estas áreas de interés se ha realizado una nueva segmentación sobre la original. Tras aplicar esta, los valores correspondientes a la clase **obstáculos** pasan a tener el valor 255, y el resto 0. De esta manera contando el número de pixeles con el valor 255 se puede comprobar si se ha detectado algún obstáculo, aplicando un umbral para establecer cuando se da como positiva la presencia de este. El caso por defecto es cuando todos los pixeles tienen el valor 0, es decir, la imagen es totalmente negra ya que no se ha encontrado ningún obstáculo en esa área. En este estado, y sin que se active ninguna de las paradas que se explican más adelante, el modelo avanzará hacia delante.

Figura 62 y 63. Vista de "Segmentation ROI" y "Collision ROI" cuando no hay obstáculos cercanos, respectivamente.

Un segundo estado es cuando se detecta la presencia de un obstáculo en el área representada en "Collision ROI", que también se encontrará en la otra vista, ya que esta segunda vista abarca a la primera. A partir de esta se inicia una corrección del rumbo, que consiste en añadir un grado de giro a la trayectoria, hasta que se deja de detectar ningún elemento en la zona que se puede observar en "Segmentation ROI". En la figura 64 se representa el momento en el que el obstáculo es detectado y este puede ser visto en ambas vista, mientras que en la figura 65 se puede observar cuando ya no se muestra este elemento en "Collision ROI", pero sigue estando en "Segmentation ROI", por lo que la corrección seguirá hasta que se deje de ver en ambas vistas.

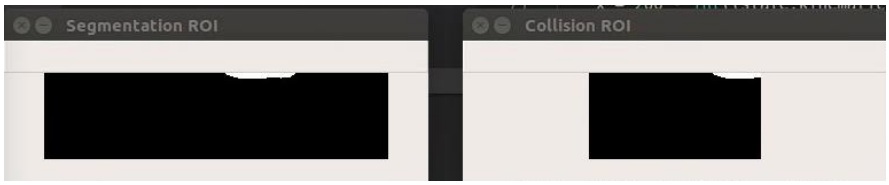


Figura 64. Vista de "Segmentation ROI" y de "Collision ROI" cuando hay un obstáculo cercano en la trayectoria actual, respectivamente.

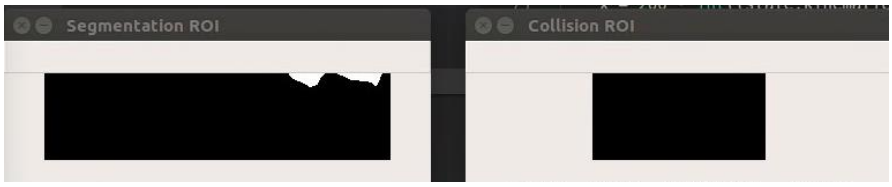


Figura 65. Vista de "Segmentation ROI" y de "Collision ROI" cuando se está esquivando un obstáculo, respectivamente.

El sentido de giro es dependiente de la segmentación que se encuentra en la área representada en "Segmentation ROI". Para determinar este se analiza los pixeles en los extremos laterales en el frame que es encontrado el obstáculo en el área de "Collision ROI" por primera vez. Si en la zona comprendida en los primeros valores del eje X se encuentra un obstáculo o varios, se esquiva estos por la derecha, en cambio si estos son encontrados en los últimos valores del eje X, se evita los obstáculos por la izquierda. Si no se encontrase ningún elemento en ninguno de los extremos de la segmentación, se realizaría por defecto por la derecha.

6.2.1.4. Vistas de las paradas de emergencia

Al igual que las vistas del subapartado anterior, "Emergency obstacle" y "Emergency impassable" muestran dos áreas de la segmentación original que han vuelto a ser segmentadas, pero en esta ocasión cada una de las vista se ha segmentado en base a una clase diferente. En el caso de la primera se busca detectar la clase **obstáculo**, por lo que la segmentación será igual a la anterior, pero en el caso de la segunda nos interesa la clase **intransitable** (o **impassable** en inglés). Es por ello que en cada una de estas vista el valor 255 representa una clase diferente, pero ambas áreas son la misma, solo cambiando la segmentación que se le aplica.

Respecto al ROI, este es el mismo en ambas vistas, por lo que ambas imágenes abarcan la misma zona de la imagen original. Siendo una región más cercana a la cámara que en "Segmentation ROI" y "Collision ROI", ya que en este caso ambas áreas son utilizadas para detectar elementos que provocan paradas de emergencia para evitar colisiones inminentes.

Al igual que en el caso anterior, el caso por defecto es cuando ningún elemento es detectado en ninguno de los dos, no interfiriendo en el avance del modelo y en la vista viéndose todos los pixeles con valor 0 y mostrándose en negro la imagen resultante de la segmentación. Siendo este el caso que se puede observar en la figura 66 y 67.

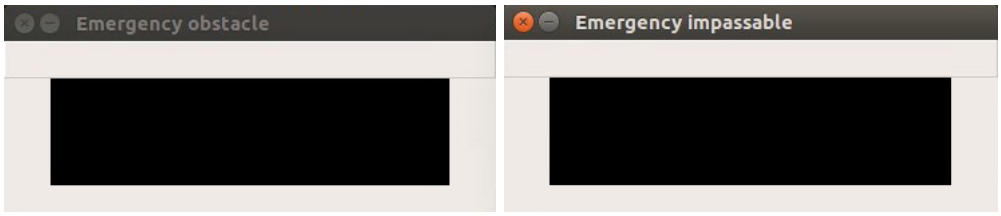


Figura 66 y 67. Vista de "Emergency obstacle" y "Emergency impassable" cuando no hay ningún elemento cerca.

Al encontrarse un elemento en el área comprendida en estas vistas, se activa una señal que indica que se debe parar inmediatamente. En el caso de que el elemento pertenezca a la clase **obstáculo**, se daría el caso que se muestra en la figura 68. En cambio, si el elemento perteneciera a la clase **intransitable**, sería como en la figura 69. Al darse esta situación, se le da prioridad sobre el resto de las señales del sistema, parándose el modelo a pesar de que se encuentre en medio de otra acción.

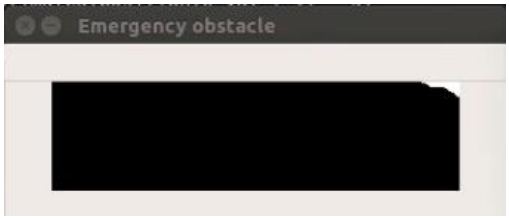


Figura 68. Vista de "Emergency obstacle" cuando hay un obstáculo cerca.

Figura 69. Vista de "Emergency impassable" cuando se encuentra un elemento de la clase "intransitable" cerca.

6.2.2. Resultados de la simulación

Para esta simulación se han realizado cinco ejecuciones con variaciones entre ellas, aunque el funcionamiento general del sistema sigue la misma estructura. Primero se comienza en un estado en el que se indica que se debe avanzar hacia delante. El modelo sigue la misma trayectoria al menos que en el área mostrada en "Collision ROI" se detecte un obstáculo. En caso de ser así, se comienza una corrección del rumbo, añadiendo grados de giro hacia la izquierda o la derecha dependiendo de la posición de los obstáculos, como se explica en el subapartado anterior. Esta corrección sigue hasta que el elemento detectado originalmente abandona el área comprendida en "Segmentation ROI". A partir de este punto empieza una corrección de rumbo, pero en este caso para volver a la posición en el eje X original. Si en medio de este proceso se encuentra nuevamente con otro obstáculo,

volverá a esquivarlo, y una vez acabado este proceso, continuará con su corrección de rumbo hacia la posición en el eje X original.

La parada del modelo se podría establecer a partir de su llegada a un posición determinada, pero en su lugar se ha decidido que esta se realice mediante el sistema de parada de emergencia implementado a partir de la segmentación en las área que se puede ver en “Emergency obstacle” y “Emergency impassable”. Esto es debido a que de esta manera se puede comprobar que estas paradas también funcionan correctamente en diferentes simulaciones.

6.2.2.1. Ejecución con un obstáculo

En la primera ejecución, de las cinco que componen esta simulación, se ha establecido un obstáculo en medio del trayecto que debe recorrer el modelo. Este es un barril naranja con franjas blancas, como se puede observar en la zona central de la imagen de la izquierda de la figura 70.

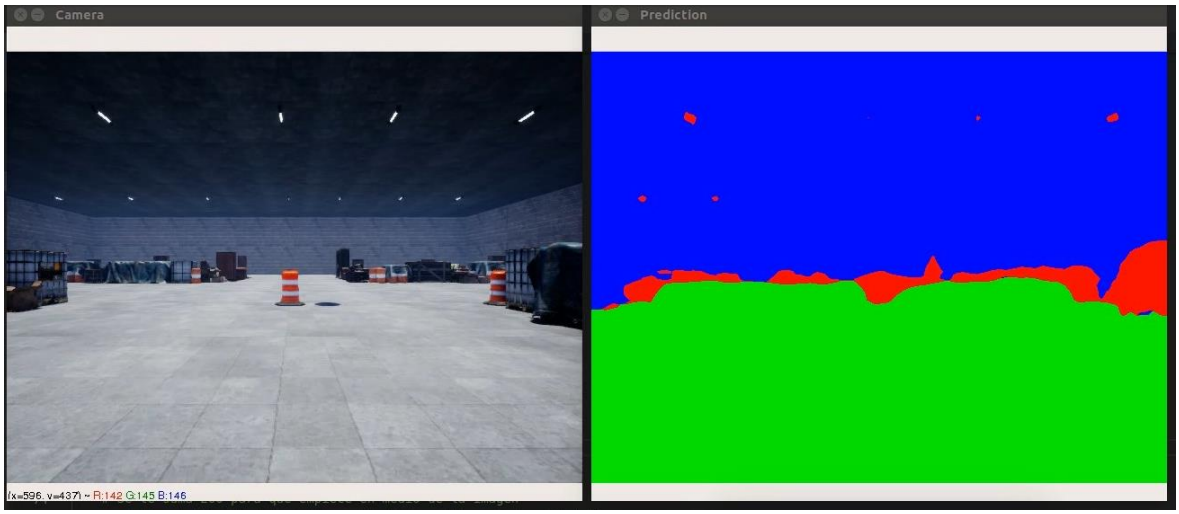


Figura 70. Escenario con un único obstáculo central.

Una vez el modelo avanza y este se va acercando al obstáculo en la vista de “Collision ROI” se detecta el barril, y se empieza una corrección de rumbo hacia la derecha, ya que al no encontrarse elementos en ninguno de los extremos de “Segmentation ROI” se utiliza este sentido por defecto. El momento a partir del cual se toma esta decisión se puede ver en la figura 71.

Figura 71. Vista de la segmentación en el área de Collision ROI y Segmentation ROI a partir de la que se decide rectificar el rumbo por la derecha.

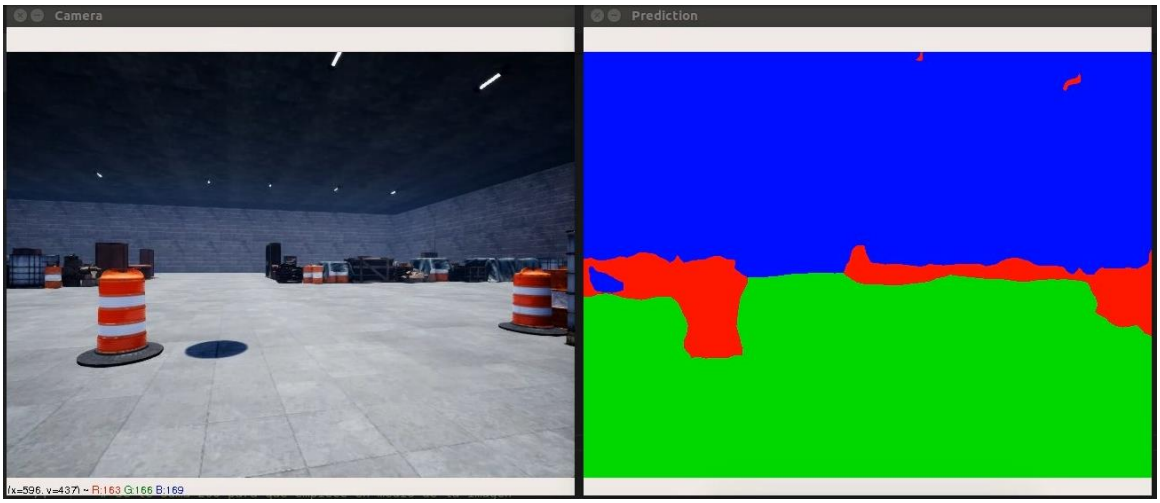


Figura 72. Giro hacia la derecha para evitar obstáculo.

En la figura 72 se puede observar cómo se realiza la corrección de rumbo hacia la derecha, siendo la sombra que se puede observar en esta producida por Unreal Engine 4 al cambiar los elementos de posición, al ser un elemento que no se calcula de forma dinámica. Si bien esta sombra no corresponde a un objeto, puede servir para comprobar que si un elemento sobrevolase la cámara su sombra no sería confundida con un objeto, mostrando que la red tiene capacidad de diferenciar entre una sombra y un obstáculo.

Tanto en la segmentación presente en la figura 72, como en la figura 70, se puede observar que existen porciones de objetos que son identificados como pertenecientes a la clase **intransitable**. A pesar de estos fragmentos donde la predicción es errónea, el resultado general es bastante correcto, por lo que el sistema puede esquivar los obstáculos de forma adecuada. La corrección de la ruta continuará hasta que se deje de observar un obstáculo en el área que se puede ver en "Segmentation ROI", estando en este punto todavía visible el barril. En la figura 73 se puede ver como el extremo de este todavía se encuentra dentro de la zona comprendida en esta vista.

Figura 73. Vista de "Segmentation ROI" y "Collision ROI" durante el proceso de cambio de trayectoria.

Una vez el obstáculo ya se ha evitado, se cambia al estado de retorno a la posición original en el eje X para continuar el trayecto que se ha definido que se debe realizar. Para ello realiza una corrección de la trayectoria en sentido contrario a la que estaba realizado hasta ahora, pasado a girar hacia la izquierda. Todo este proceso queda reflejado en el mapa generado a partir del recorrido realizado por el modelo, como se puede observar en la figura 74.

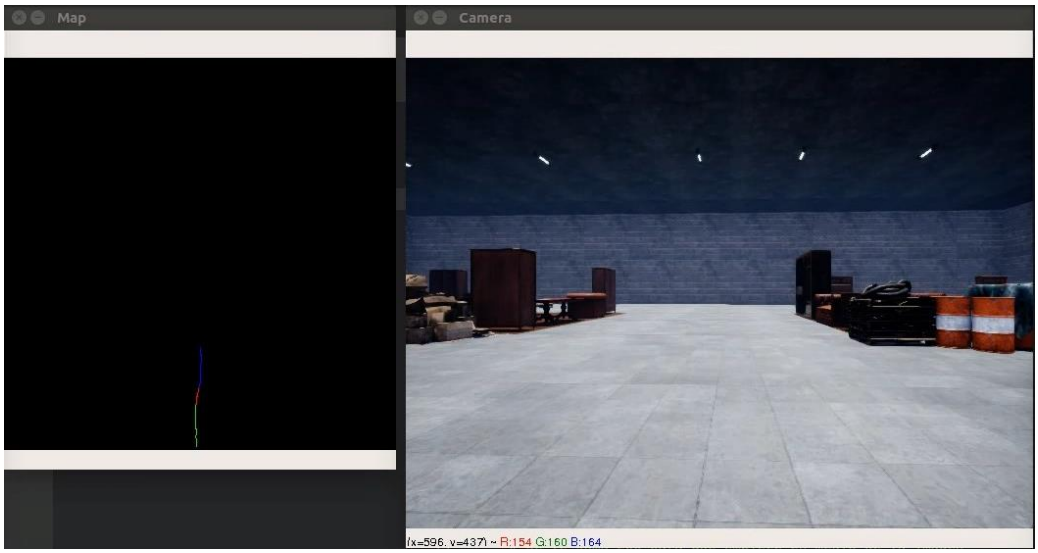


Figura 74. Vista de la cámara y del mapa durante el proceso de retorno.

Durante este proceso de retorno normalmente tanto en “Segmentation ROI” como en “Collision ROI” no se puede apreciar ningún obstáculo y todos los pixeles permanecen con valor 0, o se encuentra algún elemento, pero solo en “Segmentation ROI”, por lo que no afecta al proceso, como en el caso que se presenta en la figura 75. Pero el sistema es capaz de volver a rectificar la posición mientras está volviendo a la posición en X original, como es el caso de la situación que se puede ver en las figuras 76 y 77. Esta segunda corrección hacia la derecha se debe a que se detecta que se está aproximando demasiado a uno de los muebles que se sitúa junto al lateral del camino por el que avanza el modelo.

Figura 75. Vista de Segmentation ROI y Collision ROI durante el proceso de retorno.

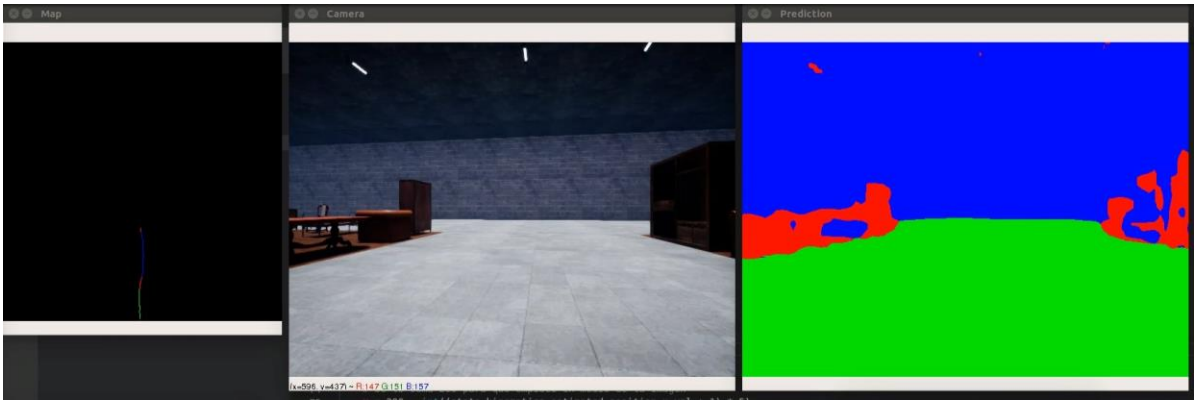


Figura 76. Corrección de rumbo durante el proceso de vuelta a la posición en X original.

Figura 77. Vista de segmentation roi y collision roi durante la segunda corrección.

Una vez se realiza esta nueva corrección del rumbo a la derecha para evitar el segundo obstáculo, se continúa volviendo a la posición en el eje X original, hasta que finalmente la alcanza y vuelve a avanzar recto. Este es el caso que se observa en la figura 78, y en el mapa que se encuentra en la parte izquierda, se puede ver como este representa todos los cambios de estados a lo largo de este proceso y como al final el camino trazado el camino trazado vuelve a tener un color verde.

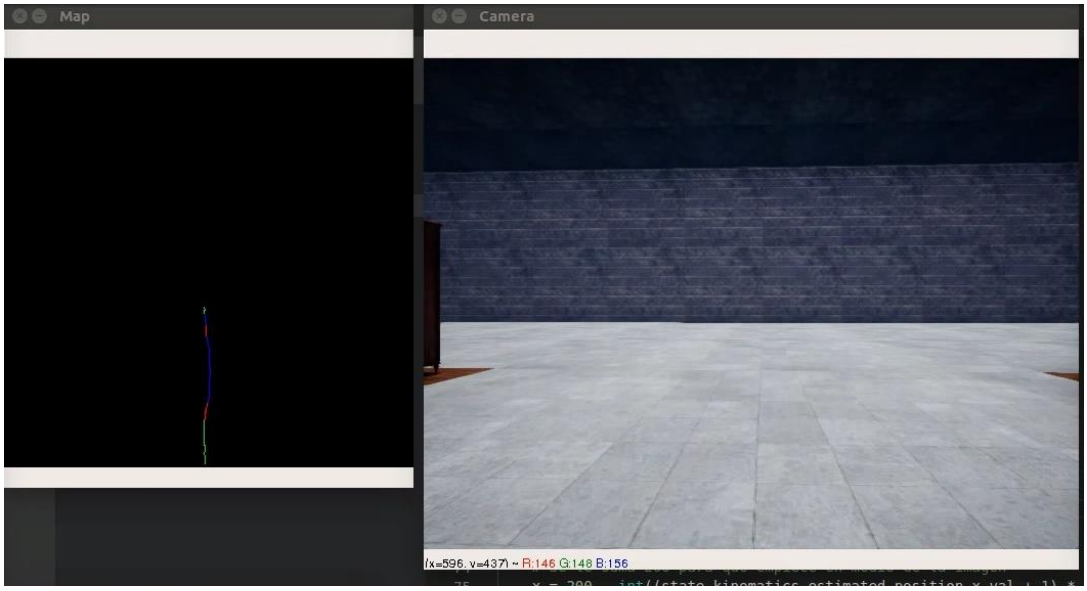


Figura 78. Vista de la cámara del modelo una vez se ha vuelto a la posición en el eje X original.

Debido a que se ha diseñado la simulación para que la parada del modelo se produzca mediante una señal de emergencia a través de la interpretación de los valores mostrados en las vistas “Emergency obstacle” y “Emergencia impassable”, el modelo sigue avanzando en la simulación hasta que finalmente se encuentra cercano a una pared que limita el escenario. El momento de esta parada se encuentra representado en la figura 79, donde se puede ver un muro cercano a la cámara.

En este caso la parada es ocasionada a partir de la señal de emergencia que se origina en base a la segmentación en la zona que representa la vista de “Emergency impassable”, donde la parte superior de la imagen tiene diversos pixeles con valor 255, representados por el color blanco, como se puede ver en la figura 80, superando el umbral permitido y activando la señal.

Figura 79. Detección final.

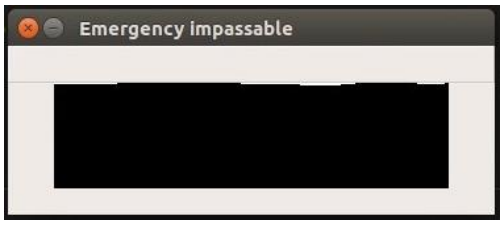


Figura 80. Vista de "Emergency impassable" en el momento de activación de la parada de emergencia.

6.2.2.2. Ejecución con dos obstáculos centrales

Con esta segunda ejecución se ha querido comprobar si se producía un cambio en la predicción o el comportamiento, para ello se ha añadido un segundo obstáculo, encontrándose este también en la ruta que debe recorrer el modelo. Ambos elementos son detectados en "Collision ROI" durante en el recorrido, como se puede ver en la figura 82, pudiéndose observar la imagen de la cámara y la segmentación de la red en este momento en la figura 81.

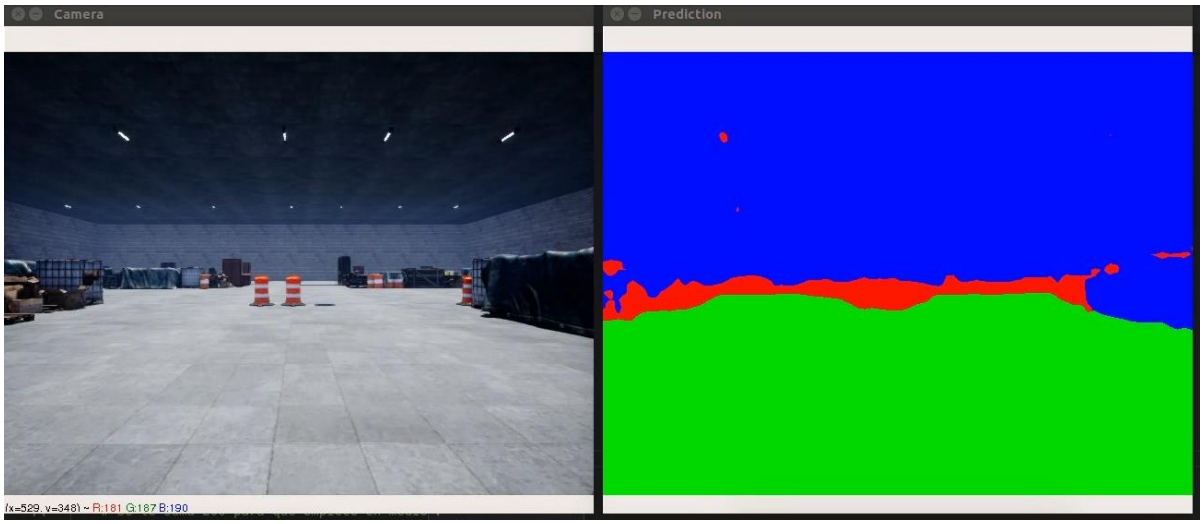


Figura 81. Escenario con dos obstáculos centrales.

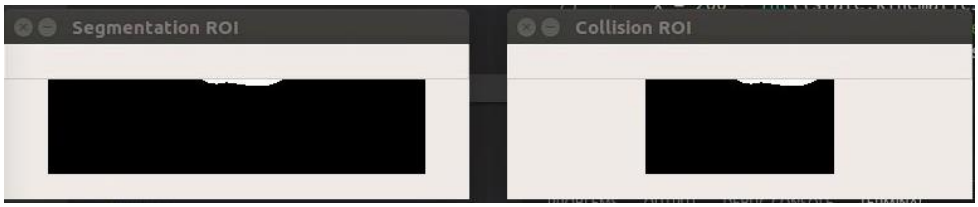


Figura 82. Detección de los obstáculos centrales.

El resultado ha sido que el conjunto de esta ejecución es similar a la del subapartado 6.2.2.1, ya que, al realizarse el giro en el mismo sentido, el cambio en el avance del modelo respecto al caso anterior es pequeño. En este caso también se ha realizado una segunda corrección del rumbo al llegar a la zona compuesta por objetos domésticos y muebles. Finalmente, al igual que en la ejecución anterior, el modelo frena al acercarse al muro que indica el final del escenario por una parada de emergencia.

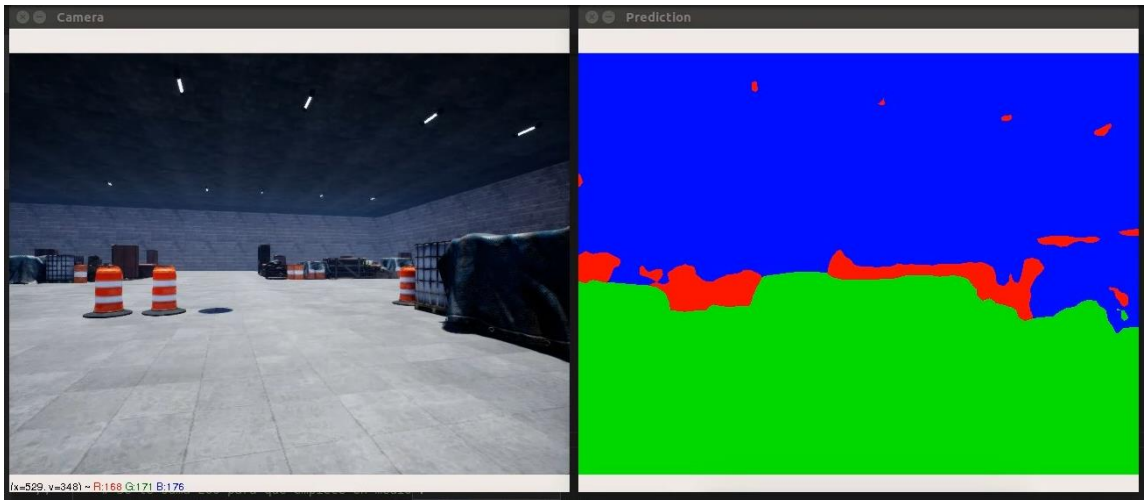


Figura 83. Corrección de la ruta hacia la derecha con los dos obstáculos a la izquierda.

Figura 84. Vista de "Segmentation ROI" y de "Collision ROI" durante el proceso de corrección de la ruta hacia la derecha.

6.2.2.3. Ejecución con un obstáculo central y uno a la derecha

En la tercera ejecución de la simulación se ha buscado, mediante la colocación de un obstáculo a la derecha del original, forzar al sistema a que a la hora de realizar la corrección del rumbo la realice mediante un giro a la izquierda, para así poder comprobar el correcto funcionamiento de la selección del sentido de giro.

En la figura 85 se puede ver la colocación de los obstáculos en esta ejecución, así como ambos son detectados en la segmentación obtenida desde ICNet. Mientras tanto en la figura 86 se representa ambas vistas a partir de las que se toma la decisión del sentido del giro, pudiéndose observar cómo en "Segmentation ROI" los pixeles con valor 255 llegan a la parte derecha de esta. En base a estos datos, el sistema decide que el mejor sentido para realizar la corrección es por la izquierda, por lo que el giro se realiza en este sentido, como se puede ver en la figura 87.

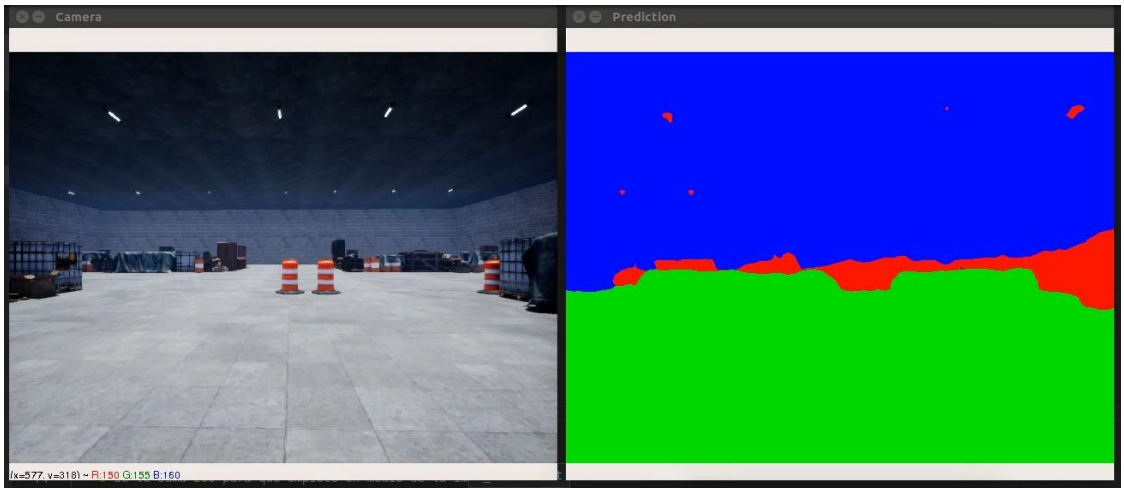


Figura 85. Escenario con un obstáculo central y otro a la derecha.

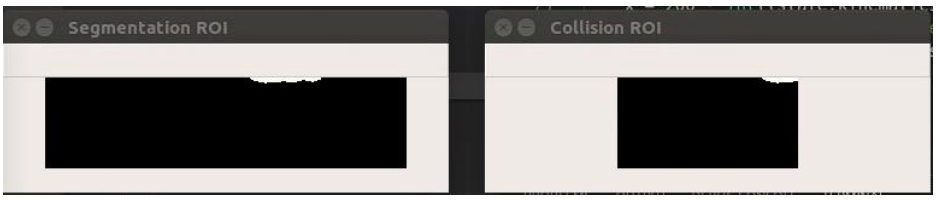


Figura 86. Vista de "Segmentation ROI" y "Collision ROI" en el momento que se decide la corrección de la ruta de realizarse hacia la izquierda.

Figura 87. Corrección de rumbo mientras se gira a la izquierda.

En esta ocasión el sistema es capaz de acabar el retorno a la posición en el eje X original sin la necesidad de realizar una segunda corrección durante este proceso, como se observa en el mapa de la figura 88. Finalmente, igual que en el resto de los casos hasta el momento, la parada se realiza por señal de emergencia en base a los datos que se muestran en la vista "Emergency impassable" al encontrarse el modelo cercano al muro que delimita el final del escenario.

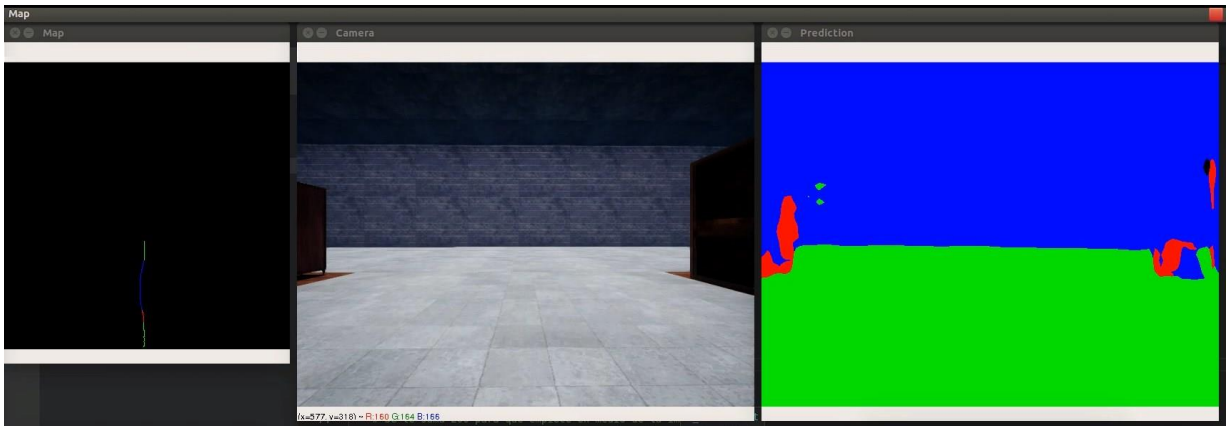


Figura 88. Mapa, vista de la cámara y segmentación realizada por ICNet una vez el rumbo ha sido corregido.

6.2.2.4. Ejecuciones con obstáculos realizando un bloqueo completo

En este caso se han realizado dos ejecuciones con el mismo escenario y los mismos obstáculos, dispuestos como se muestra en la figura 89. En el primer caso se ha buscado conseguir una parada de emergencia en base a la cercanía de un obstáculo, mientras que en el otro se ha buscado que el sistema al comprobar que no existía ningún camino posible realizase la parada de emergencia directamente. Para comprobar que ambos comportamientos tienen el funcionamiento esperado, en la primera ejecución se ha desactivado la opción de predecir que no existe un camino posible, activando está en la siguiente prueba.

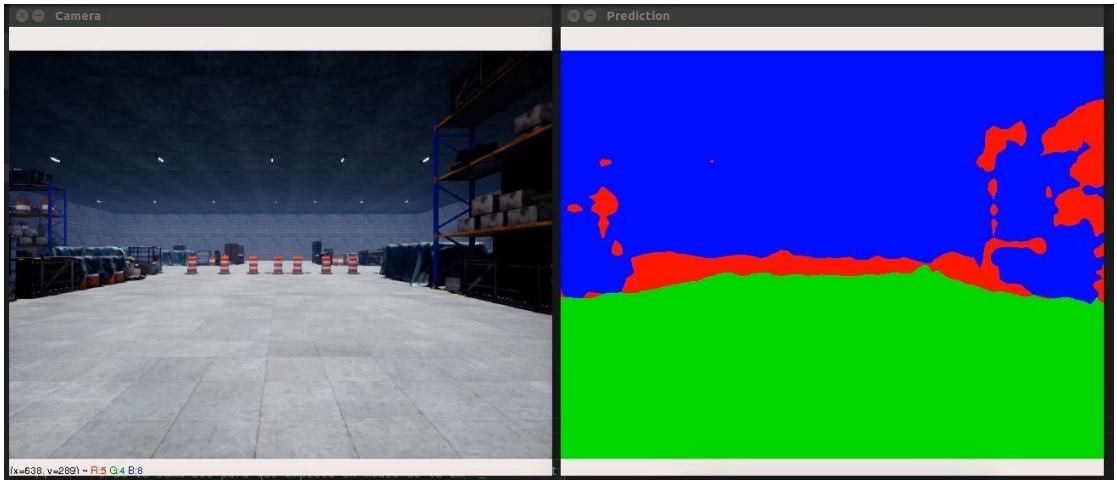


Figura 89. Escenario con un conjunto de obstáculos cortando todo camino posible.

Al estar activada la función que comprueba la posibilidad de encontrar un camino libre se realiza una comprobación extra, que consiste en comprobar si en el área comprendida en la vista “Segmentation ROI” existen obstáculos tanto en la parte central como en ambos extremos, si en los tres casos fuera el caso positivo, se realizaría la parada del modelo. Si esta opción se encontrase desactivada en su lugar se realizaría un giro a la derecha en esta situación, para intentar encontrar una posible vía que no se viese en ese momento, impidiéndose la colisión por las paradas de emergencia.

En la primera ejecución, el modelo al aproximarse a los obstáculos ha empezado a realizar una corrección de rumbo girando a la derecha, mientras sigue acercándose a los obstáculos, como se puede ver en la figura 90. Finalmente, el modelo acaba encontrándose con un obstáculo demasiado cercano, momento ilustrado en la figura 91. En este instante en el área mostrada en la vista “Emergency obstacle”, que se muestra en la figura 92, se puede comprobar la existencia de un conjunto de pixeles en blanco que indican que se debe realizar la parada de emergencia para evitar una colisión inminente.

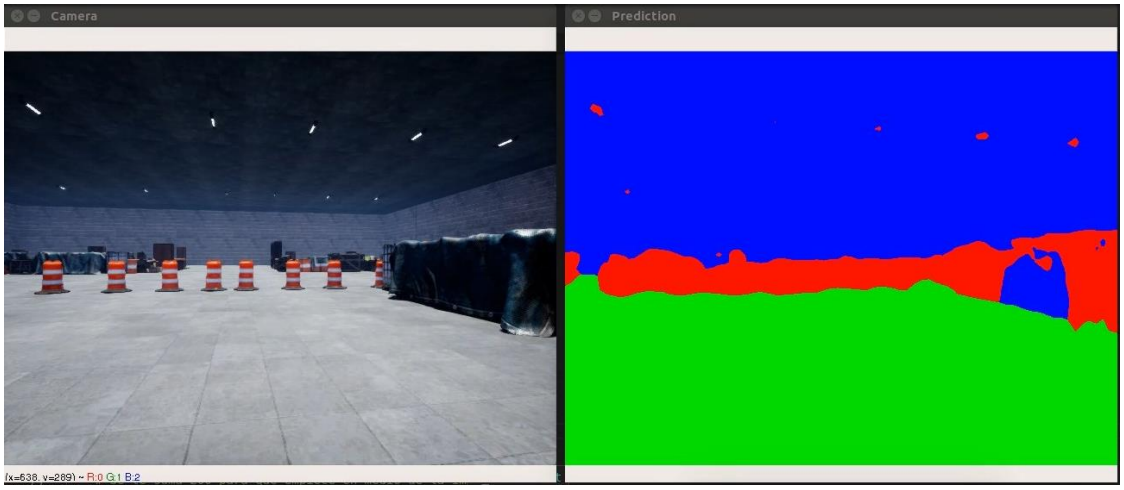


Figura 90. Corrección hacia la derecha en un escenario sin la posibilidad de evitar los obstáculos.

Figura 91. Momento de la parada del modelo debido a la proximidad a un obstáculo.

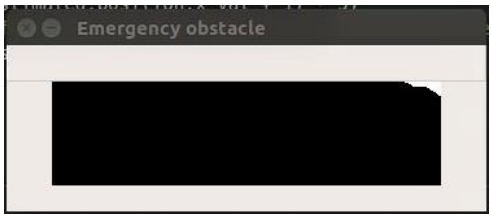


Figura 92. Vista de "Emergency obstacle" en el momento en el que se realiza la parada.

En la quinta y última ejecución se ha activado la función de parada en base a la predicción de posibles caminos, en este caso el modelo se queda a una distancia considerable de los obstáculos, como se puede ver en la figura 93, ya que es en este punto cuando comprueba que no es posible encontrar un camino para evitar los obstáculos.

En la figuras 94 se puede observar la vista de "Segmentation ROI" en el momento de la parada, comprobándose que se cumple el requisito de la presencia de obstáculos en las tres parte en que está dividido este, para que así se active la parada. Pero a pesar de ello, se puede ver que en la parte derecha del ROI apenas hay suficientes pixeles en blanco para superar el umbral mínimo necesario para que se considere como positiva la presencia de un obstáculo. Esto se debe tener en cuenta, ya que la función para predecir si existe un posible camino puede fallar en diversas situaciones por no llegar a este umbral una de las partes. Este hecho refleja que, si bien los obstáculos se han colocado en la misma posición del eje Y en el escenario, la segmentación no tiene una precisión suficiente en este caso, por lo que no se ve reflejada con exactitud la correlación en el eje Y de los obstáculo. Esto provoca que existan limitaciones a la hora de desarrollar funciones que se basen en la segmentación semántica cuando la relación entre la posición de los elementos sea muy importante.

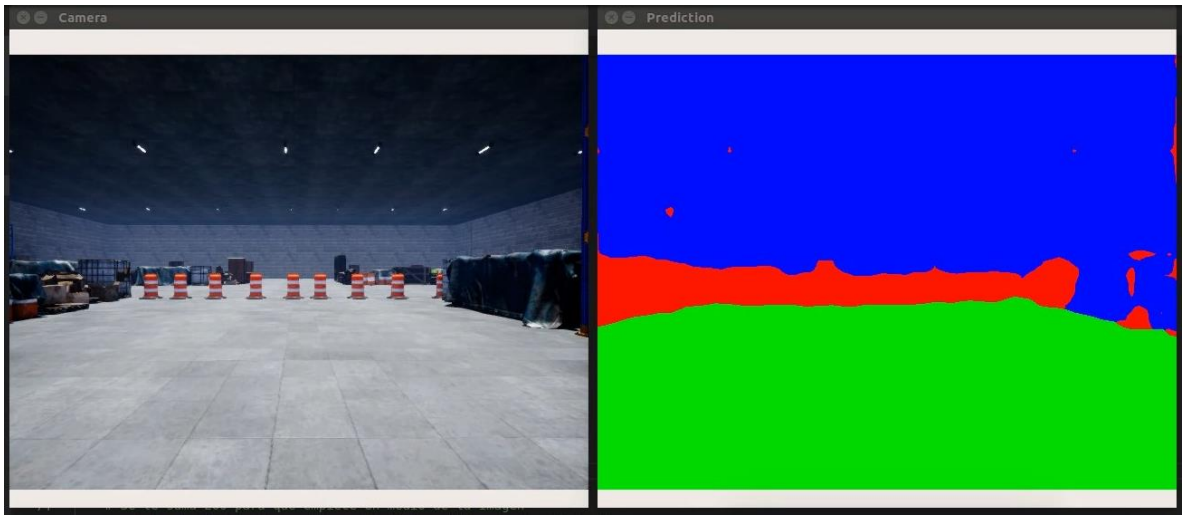


Figura 93. Vista desde la cámara del modelo en el momento en que se ocasiona la parada.

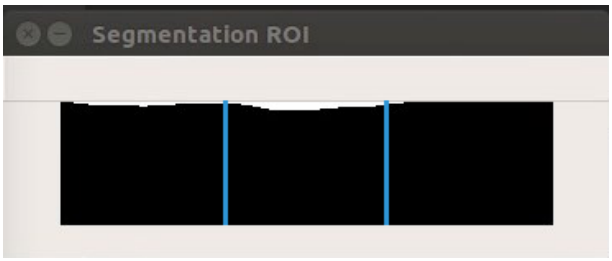


Figura 94. Vista de "Segmentation ROI" en el momento de tiene lugar la parada, con líneas en azul para distinguir las partes en la que ha sido divido para la predicción.

6.2.3. Análisis de los resultados de la simulación

El resultado de la simulación ha sido satisfactorio, ya que se ha conseguido un sistema que permite sin modificar parámetros, a excepción de la activación o desactivación de la parada por predicción de caminos posibles, evitar los obstáculos, que varían entre las cinco ejecuciones. Aunque se es consciente de que se trata de una simulación simple, y que además la colocación de los obstáculos en ciertas posiciones puede llevar a que el sistema no funcione correctamente, debido a su sencillez y a no contemplar excepciones o casos más complejos. A pesar de ello, los resultados se siguen considerando positivos en cuanto la simulación sirve como muestra de un sistema que hace uso de la segmentación semántica para evitar obstáculos, y colisiones en general, de forma satisfactoria.

7. Conclusiones

7.1. Conclusiones

En este proyecto se ha explicado a nivel teórico desde las bases del deep learning hasta el estado del arte de la segmentación semántica mediante redes neuronales profundas. También se han explorado las diferentes alternativas existentes en el momento de realización de este trabajo, comparando su rendimiento y precisión, buscando el mejor equilibrio entre ambos elementos.

Una vez establecida la red a utilizar, se ha buscado una implementación sobre la que trabajar y se ha expuesto el software que se ha utilizado en este proyecto. Con estos fundamentos establecidos, se ha trabajado sobre una implementación de ICNet y se ha mejorado esta hasta obtener unos resultados satisfactorios, realizando diversas pruebas para comprobarlos.

Todo este trabajo ha servido para definir una buena red para conseguir el propósito establecido y poder realizar un conjunto de pruebas que han permitido saber el rendimiento que podemos obtener en un hardware de coste moderado. Se han conseguido los objetivos establecidos tanto en lo referente al coste del hardware como en lo referente al rendimiento, ya que se ha conseguido ejecuta a más de 30fps la red en una GPU de menos de 400€, dejando un margen de 600€ para el resto del hardware.

Finalmente, se ha puesto en práctica todo el trabajo realizado en una simulación en Unreal Engine 4, donde se ha simulado un escenario tanto con elementos domésticos como con elementos de carácter industrial. En esta se ha implementado un sistema que permitía que el modelo que se controlaba esquivase obstáculos y corrigiese su rumbo en base a la segmentación semántica obtenida por ICNet.

Si bien la simulación realizada presenta un caso de uso simple, y no se puede extrapolar directamente a situaciones reales, sí que se considera que demuestra parte del potencial que tiene la segmentación semántica, mediante deep learning, para la navegación en interiores, dejando vislumbrar los posibles usos que se le puede dar en el futuro.

7.2. Futuras líneas de investigación

Las posibles maneras en las que este proyecto se puede continuar, ampliar y mejorar son diversas, ya que es un campo donde hay muchas posibles mejoras y diferentes opciones a desarrollar. Pero entre estas opciones, las siguientes son las que se consideran más importante de cara a la ampliación de este trabajo:

- Estudio de implementación y consumo
- Optimización del software y del hardware

7.2.1. Estudio de implementación y consumo

En este proyecto se investiga el uso de la segmentación semántica para la navegación en interiores, pero no se ha podido llegar a realizar la implementación de esta navegación en un robot real.

Esta implementación podría realizarse en forma de ampliación de este trabajo, permitiendo estudiar la importancia de diferentes elementos a la hora de realizarla, como son la importancia del consumo del hardware y el tamaño de este. Además, de estudiar el resto del hardware necesario para el procesamiento, como puede ser la CPU o el almacenamiento, en vez de únicamente centrarse en la GPU.

También permitiría comprobar en un escenario real que funciones se podrían realizar únicamente con segmentación semántica, así como aquellas en las que se podría utilizar, pero como un elemento auxiliar para mejorar otros sistemas ya existentes.

7.2.2. Optimización del software y del hardware

Desde el punto de vista del software existen diferentes opciones a explorar para la optimización del rendimiento del algoritmo. Una de las opciones que más destaca es TensorRT, ya que puede combinarse con TensorFlow para reconstruir el grafo de forma más optima.

Esta librería, dependiendo del hardware, da la posibilidad a utilizar diferentes niveles de precisión, pudiéndose utilizar floats de 16 bits de precisión y enteros sin signo de 8 bits, en vez de utilizar los floats de 32 bits de precisión que se utilizan normalmente. Esto abre la posibilidad a poder optimizar el grafo y poder utilizar hardware que requiera menos potencia o a realizar underclocking al hardware que se ha utilizado, en busca de un menor consumo si fuera necesario.

Desde el punto de vista del hardware, hay que tener en cuenta el coste y el consumo. El hardware utilizado para las pruebas se trata de diferentes GPUs de NVIDIA, ya que permiten utilizar CUDA. Pero si bien este hardware permite utilizar ICNet llegando al umbral de los 30FPS con un coste moderado, hay que contemplar escenarios donde este hardware represente un exceso de consumo eléctrico o de coste. Por ello, se deben explorar opciones que reduzcan ambos factores.

En el apartado de TPUs, se debería explorar el uso de otras tecnologías como Intel Movidius Myriad 2, que aporta un menor consumo, aunque tenga menor potencia y mayores limitaciones que las GPUs utilizadas. También se podría considerar el uso del SBC Google Dev Board que incorpora una TPU con soporte para TensorFlow a un coste de 149$.

Además, también se debería explorar el uso de hardware que ha visto la luz durante el desarrollo de este trabajo, como Jetson Nano. Que combina el uso de CUDA y una GPU en un SBC por 99$, teniendo soporte para TensorRT.

Todas las alternativas citadas anteriormente en este subapartado podrían suponer un salto cualitativo en este proyecto, al combinar la reducción de consumo, coste y tamaño.

7.3. Conclusiones personales

A nivel personal este trabajo me ha permitido entender los fundamentos del deep learning, avanzando a través del estado del arte de este, observando así su evolución y entendiendo como se ha ido desarrollando. Considero que este trabajo me ha resultado realmente útil, y que en el futuro me servirá como base para poder trabajar con redes neuronales.

A nivel del trabajo desarrollado, me he encontrado con que la carga teórica ha superado a la cantidad de trabajo práctico a desarrollar, en mayor medida a lo esperado. Si bien considero útil todo el trabajo teórico realizado, a través de la búsqueda de información y realización de cursos online, este ha supuesto un reto, teniendo que cambiar la planificación realizada inicialmente para darle un mayor peso al trabajo teórico.

Bibliografía

[1] Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep Learning*. MIT Press, pp.1-26. Disponible en: https://www.deeplearningbook.org/

[2] Torres, J. (2016). *Hello world en tensorflow - para iniciarse en la programacion del deep learning*. LULU. Disponible en: https://torres.ai/es/tensorflow/libro-hello-world-en-tensorflow/

[3] McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), pp.115-133.

[4] Fountas, Z. (2011). Spiking Neural Networks for Human-like Avatar Control in a Simulated Environment. *Imperial College London, Department of Computing*.

[5] Cs231n.github.io. (2019). *CS231n Convolutional Neural Networks for Visual Recognition*. [online] Disponible en: http://cs231n.github.io/convolutional-networks/ [Accedido 10 Feb. 2019].

[6] Miriam Mónica Duarte Villaseñor y Leonardo Chang Fernández. Clasificación de objetos en imágenes usando SIFT. *Proyecto del Curso Modelos Gráficos Probabilistas y sus aplicaciones Maestría en Ciencias Computacionales, INAOE*. Disponible en: http://ccc.inaoep.mx/~esucar/Clases-mgp/Proyectos/chang-duarte.pdf

[7] Amy R. Wagoner, Daniel K. Schrader, and Eric T. Matson. *Survey on Detection and Tracking of UAVs Using Computer Vision.* 2017 First IEEE International Conference on Robotic Computing (IRC), Taichung, 2017 pp. 320-325. Disponible en: https://www.computer.org/csdl/proceedings/irc/2017/6724/00/07926557.pdf

[8] Ogrisel.github.io. (2019). *Tutorial Diagrams — scikit-learn 0.11-git documentation*. [online] Disponible en: http://ogrisel.github.io/scikit-learn.org/sklearn-tutorial/auto_examples/tutorial/plot_ML_flow_chart.html [Accedido 3 Mar. 2019].

[9] Dertat, A. (2019). *Applied Deep Learning - Part 1: Artificial Neural Networks*. [online] Towards Data Science. Disponible en: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6 [Accedido 3 Mar. 2019].

[10] Explained Visually. (2019). *Image Kernels explained visually*. [online] Disponible en: http://setosa.io/ev/image-kernels/ [Accedido 6 Mar. 2019].

[11] Keras Visualization Toolkit. (2019). *raghakot/keras-vis*. [online] Disponible en: https://github.com/raghakot/keras-vis [Accedido 10 Mar. 2019].

[12] Sharma, S. (2019). *Activation Functions in Neural Networks*. [online] Towards Data Science. Disponible en: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 [Accedido 13 Mar. 2019].

[13] Singh Walia, A. (2019). *Activation functions and it's types-Which is better?*. [online] Towards Data Science. Disponible en: https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f [Accedido 17 Mar. 2019].

[14] Xu, B., Wang, N., Chen, T., & Li, M. (2015). *Empirical Evaluation of Rectified Activations in ConvolutionNetwork.* ArXiv. Disponible en: https://arxiv.org/pdf/1505.00853.pdf

[15] Jed-ai.github.io. (2019). *JedAI - 7 Simple Steps To Visualize And Animate The Gradient Descent Algorithm*. [online] Disponible en: https://jed-ai.github.io/py1_gd_animation/ [Accedido 19 Mar. 2019].

[16] Bogotobogo.com. (2019). *scikit-learn: Batch gradient descent versus stochastic gradient descent - 2018*. [online] Disponible en: https://www.bogotobogo.com/python/scikit-learn/scikit-learn_batch-gradient-descent-versus-stochastic-gradient-descent.php [Accedido 20 Mar. 2019].

[17] Towards Data Science. (2019). *Convolutional Neural Network*. [online] Disponible en: https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05 [Accedido 5 Abr. 2019].

[18] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. (2014). *Dropout: A Simple Way to Prevent Neural Networks fromOverfitting.* Journal of Machine Learning Research 15 (2014), pp.1929-1958.

[19] Bhande, A. (2019). *What is underfitting and overfitting in machine learning and how to deal with it*. [online] Medium. Disponible en: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76 [Accedido 9 Abr. 2019].

[20] Y.Ng, A. (2004). *Feature selection, L1 vs. L2 regularization, and rotational invariance.* ICML '04 Proceedings of the twenty-first international conference on Machine learning, p. 6.

[21] Deshpande, A. (2019). *A Beginner's Guide To Understanding Convolutional Neural Networks*. [online] Adeshpande3.github.io. Disponible en: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/ [Accedido 10 Abr. 2019].

[22] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. (2019). *You Only Look Once: Unified, Real-Time Object Detection*. University of Washington, Allen Institute for AI, Facebook AI Research, p.2.

[23] Badrinarayanan, V., Kendall, A. and Cipolla, R. (2016). *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. arXiv, p. 4. Disponible en: https://arxiv.org/pdf/1511.00561.pdf

[24] Zhao, H., Qi, X., Shen, X., Shi, J. and Jia, J. (2019). *ICNet for Real-Time Semantic Segmentation on High-Resolution Images*. [online] The Chinese University of Hong Kong, Tencent Youtu Lab, SenseTime Research. Disponible en: https://arxiv.org/pdf/1704.08545.pdf [Accedido 10 Abr. 2019].

[25] Zawadzki, J. (2019). *The Deep Learning(.ai) Dictionary*. [online] Towards Data Science. Disponible en: https://towardsdatascience.com/the-deep-learning-ai-dictionary-ade421df39e4 [Accedido 1 Mayo 2019].

[26] Simonyan, K. and Zisserman, A. (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. [online] arXiv. Disponible en: https://arxiv.org/pdf/1409.1556.pdf [Accedido 4 Mayo 2019].

[27] Higham, D. (2019). *How we use image semantic segmentation*. [online] Medium. Disponible en: https://medium.com/digitalbridge/how-we-use-image-semantic-segmentation-e85fac734caf [Accedido 5 Mayo 2019].

[28] Long, J., Shelhamer, E. and Darrell, T. (2019). *Fully Convolutional Networks for Semantic Segmentation*. [online] Disponible en: https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf [Accedido 5 Mayo 2019].

[29] Python.org. (2019). *What is Python? Executive Summary*. [online] Disponible en: https://www.python.org/doc/essays/blurb/ [Accedido 19 Jun. 2019].

[30] Docs.scipy.org. (2019). *What is NumPy? — NumPy v1.13 Manual*. [online] Disponible en: https://docs.scipy.org/doc/numpy-1.13.0/user/whatisnumpy.html [Accedido 20 Jun. 2019].

[31] Notes-on-cython.readthedocs.io. (2019). *The Performance of Python, Cython and C on a Vector — Cython def, cdef and cpdef functions 0.1.0 documentation*. [online] Disponible en: https://notes-on-cython.readthedocs.io/en/latest/std_dev.html [Accedido 20 Jun. 2019].

[32] Tantamjarik, E. (2019). *CUDA Basic Concept - Eakawat Tantamjarik - Medium*. [online] Medium. Disponible en: https://medium.com/@eakawattantamjarik/cuda-basic-concept-740cdd6f984 [Accedido 20 Jun. 2019].

[33] Zhao, H., Shi, J., Qi, X., Wang, X. and Jia, J. (2017). *Pyramid Scene Parsing Network*. [online] The Chinese University of Hong Kong. Disponible en: https://arxiv.org/abs/1612.01105# [Accedido 20 Jun. 2019].

[34] NVIDIA Developer. (2019). *NVIDIA cuDNN*. [online] Disponible en: https://developer.nvidia.com/cudnn [Accedido 29 Jun. 2019].

[35] Opencv.org. (2019). *About*. [online] Disponible en: https://opencv.org/about/ [Accedido 29 Jun. 2019].

[36] Opensource.com. (2019). *What is the TensorFlow machine intelligence platform?*. [online] Disponible en: https://opensource.com/article/17/11/intro-tensorflow [Accedido 29 Jun. 2019].

[37] Gudivada, V. and Arbabifard, K. (2019). Open-Source Libraries, Application Frameworks, and Workflow Systems for NLP. *Handbook of Statistics*, [online] 38, p.36. Disponible: https://www.researchgate.net/publication/326885514_Open-Source_Libraries_Application_Frameworks_and_Workflow_Systems_for_NLP [Accedido 1 Jul. 2019].

[38] GitHub. (2019). *bulletphysics/bullet3*. [online] Disponible en: https://github.com/bulletphysics/bullet3 [Accedido 7 Jul. 2019].

[39] Virtualenv.pypa.io. (2019). *Virtualenv — virtualenv 16.6.1 documentation*. [online] Disponible en: https://virtualenv.pypa.io/en/latest/ [Accedido 7 Jul. 2019].

[40] Tsang, S. (2019). *Review: DilatedNet — Dilated Convolution (Semantic Segmentation)*. [online] Medium. Disponible en: https://towardsdatascience.com/review-dilated-convolution-semantic-segmentation-9d5a5bd768f5 [Accedido 21 Jul. 2019].

[41] Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S. and Schiele, B. (2016). *The Cityscapes Dataset for Semantic Urban Scene Understanding*. [online] Disponible en: https://arxiv.org/pdf/1604.01685.pdf [Accedido 22 Jul. 2019].

[42] GitHub. (2019). *hellochick/ICNet-tensorflow*. [online] Disponible en: https://github.com/hellochick/ICNet-tensorflow [Accedido 24 Oct. 2018].

[43] Cityscapes-dataset.com. (2019). *Dataset Overview – Cityscapes Dataset*. [online] Disponible en: https://www.cityscapes-dataset.com/dataset-overview/ [Accedido 25 Jul. 2019].

[44] Zhou, B., Zhao, H., Puig, X., Fidler, S., Barriuso, A. and Torralba, A. (2017). *Scene Parsing through ADE20K Dataset*. [online] Disponible en: http://people.csail.mit.edu/bzhou/publication/scene-parse-camera-ready.pdf [Accedido 25 Jul. 2019].

[45] Paszke, A., Chaurasia, A., Kim, S. and Culurciello, E. (2016). *Enet: A deep neural network architecture for real-time semantic segmentation*. [online] Disponible en: https://arxiv.org/pdf/1606.02147.pdf [Accedido 26 Jul. 2019].

[46] Chen, L., Papandreou, G., Kokkinos, I., Murphy, K. and Yuille, A. (2019). *Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs*. [online] Disponible en: https://arxiv.org/pdf/1606.00915.pdf [Accedido 26 Jul. 2019].

[47] TensorFlow. (2019). *TensorBoard: Visualizing Learning | TensorFlow Core | TensorFlow*. [online] Disponible en: https://www.tensorflow.org/guide/summaries_and_tensorboard [Accedido 4 Agosto 2019].